

Proceedings of the 19th Central European Seminar on Computer Graphics

April 20 - 22, 2015
Smolenice, Slovakia
Co-organized with SCCG



Institute of Computer Graphics and Algorithms
Vienna University of Technology

Spring
conference
on computer
graphics



Faculty of Mathematics, Physics and Informatics
Comenius University Bratislava



Sponsors

• Visegrad Fund



Slovak Society of
Computer Science



OESTERREICHISCHE
COMPUTER GESELLSCHAFT
AUSTRIAN
COMPUTER SOCIETY

Impressum

Vienna University of Technology
Institute of Computer Graphics and Algorithms
Favoritenstraße 9-11/186
1040 Vienna

ISBN 978-3-9502533-7-5

Welcome to CESC 2015!

This book contains the proceedings of the 19th Central European Seminar on Computer Graphics, short CESC, which continues a history of very successful seminars. Again this year, CESC proceedings have an ISBN (978-3-9502533-7-5) and will therefore remain retrievable as long as there are libraries!

The long history of CESC has started in 1997 in a medium-sized lecture room in Bratislava, bringing together students from Bratislava, Brno, Budapest, Graz, Prague, and Vienna. The idea found wide appraisal and the seminar moved to the beautiful castle of Budmerice, where it was held for 8 consecutive years, constantly growing in size and attraction. It was just in the 10th anniversary year 2006 that CESC had to take a detour to move to Častá-Papiernička Centre, while it was back in Budmerice castle in 2007. Unfortunately, since 2011 the Budmerice castle is not available for scientific activities. After spending the one year in Viničné, in 2012 we moved to the beautiful castle in Smolenice.

Who are the CESC heroes who made this year's seminar happen? In no particular order – because many people were involved equally – we would like to thank the organizers from Vienna: **Michael Wimmer**, Anita Mayerhofer, Katharina Krösl, and Werner Purgathofer. Special thanks goes to **Martin Ilčík** for taking care of the complete reviewing process and scientific program preparation. We are very thankful to the CESC organizers from Bratislava, mainly **Andrej Ferko**, always an inspiration to CESC; Ela Šikudová, and Michal Ferko for the excellent preparations and on-site organization.

The main idea of CESC is to bring students of computer graphics together across boundaries of universities and countries. We mainly focus on sustainable academic and research development in the field of Computer Graphics in Visegrad Countries and Austria. Our mission is to support undergraduate talents in their future careers. Therefore, we are proud to state that we have achieved again a very high number of 19 participating institutions and a very tight time schedule of 21 valuable student works, 3 specialized interactive workshops, a discussion panel and two invited talks. We welcome groups from Bratislava (UK and STU), Slovakia; Brno (VUT and MU) and Prague (CTU), Czech Republic; Budapest (BME), Hungary; Bonn, Germany; Graz and Vienna (TU and VRVis), Austria; Szczecin, Poland; Maribor, Slovenia; and Sarajevo (UnSa), Bosnia and Herzegovina.

We assembled an International Program Committee of 13 members, allowing us to have each paper reviewed by three IPC members during the informal reviewing process. We would like to thank the members of the IPC for their contribution to the reviewing process. The IPC of CESC 2015 consists of:

Jiří Bittner
Andrej Ferko
Ivana Kolingerová
Radosław Mantiuk
Selma Rizvić
Michael Schwärzler
Pavel Zemčík

Jiří Sochor
Markus Steinberger
László Szirmay-Kalos
Ania Tomaszewska
Michael Wimmer
Borut Žalik

The reviewing process was further supported by: Thomas Auzinger, Zuzana Berger Haladová, Michael Birsak, Pedro Boechat, Peter Borovský, Pavel Chalmovianský, Daniel Cornel, Roman Ďurikovič, Pavol Fabo, Michal Ferko, Michael Hecher, Michael Kenzel, Christian Luksch, Przemysław Musialski, Stefan Ohrhallinger, Reinhold Preiner, Mohamed Radwan, Elena Šikudová, Stanislav Stanek, Bernhard Steiner, Ivana Uhlíková, Ivana Varhaníková, and Károly Zsolnai.

The first invited talk “Computational Challenges in Designing Virtual Models for Fabrication” will be held by Bernd Bickel from the Institute of Science and Technology, Austria. The second invited talk by Tamás Várady from the Department of Control Engineering and Information Technology of the Budapest University of Technology and Economics, Hungary, will be about “Challenges in Digital Shape Reconstruction”. The workshop “What makes a great talk?” will be held by Károly Zsolnai. Martin Ilčík will guide the students in the second workshop called “The Hero Journey in Science”.

The seminar co-organized with the Spring Conference on Computer Graphics (SCCG), which takes place right after. For the first time we created an overlapping day with a joint program. On Wednesday morning a panel on “Computer Graphics Education in Visegrad Countries” will be hosted by Dr. Michael Wimmer. In the afternoon, four CESCg papers which managed to be promoted to the SCCG will be presented:

Marek Galváněk: Automated Facial Landmark Detection, Comparison and Visualization

Márton Vaitkus: A General Framework for Constrained Mesh Parameterization

Gergely Rácz: Tomographic Reconstruction on the Body-Centered Cubic Lattice

Martin Sattlecker: Reyes Rendering on the GPU

Please refer to the SCCG proceedings for the full-text of the promoted papers.

The organization of a seminar where there are only low expenses for the students requires funding. We are very thankful to the sponsors of CESCg 2015:

- International Visegrad Fund, <http://visegradfund.org>,
- NVidia, The Way It’s Meant to Be Played,
- VRVis, Research Center for Virtual Reality and Visualization,
- OCG, The Austrian Computer Association,
- SISp, Slovak Society for Computer Science,
- Eurographics, The European Association for Computer Graphics.

Please note that the electronic version of these proceedings is also available at <http://www.cescg.org/CESCg-2015/>.

April 2015,

Michael Wimmer
Jiří Hladůvka
Martin Ilčík

Table of Contents

Invited Talks

Computational Challenges in Designing Virtual Models for Fabrication	3
<i>Bernd Bickel. Institute of Science and Technology, Austria</i>	
Challenges in Digital Shape Reconstruction	5
<i>Tamás Várady. Budapest University of Technology and Economics, Hungary</i>	

GPU

Accelerated gSLIC for Superpixel Generation used in Object Segmentation	9
<i>Robert Birkus. Slovak University of Technology, Slovakia</i>	
Real-time Water Simulation with Wave Particles on the GPU	17
<i>Daniel Mikeš. Czech Technical University, Czech Republic</i>	
Arbitrary-Precision Arithmetics on the GPU	25
<i>Bernhard Langer. Vienna University of Technology, Austria</i>	

Visualization & Applications

OpenGL View Library	35
<i>Adam Riečický. Comenius University, Slovakia</i>	
Visualizing Archaeological Excavations based on Unity3D	41
<i>Thomas Trautner. VRVis, Austria</i>	
Unity Hyperlinked Interactive Digital Storytelling	49
<i>Irfan Prazina. University of Sarajevo, Bosnia and Herzegovina</i>	
CellUnity - an Interactive Tool for Illustrative Visualization of Molecular Reactions	57
<i>Daniel Gehrler. Vienna University of Technology, Austria</i>	

Real-Time Rendering & Illumination

Fast photon gathering in progressive photon mapping using GPGPU	67
<i>Tomáš Lysek. Brno University of Technology, Czech Republic</i>	
Efficient Implementation of Bi-directional Path Tracer on GPU	75
<i>Vilém Otte. Masaryk University, Czech Republic</i>	
Rendering Large Point Clouds in Web Browsers	83
<i>Markus Schütz. Vienna University of Technology, Austria</i>	
Order Independent Transparency with Non-local Opacity Modulation for 3D Meshes	91
<i>Tomáš Pastýřík. Czech Technical University, Czech Republic</i>	

Vision & Perception

Fast Detection of the Pupil Centre in Stable Light Conditions	101
<i>Krzysztof Wolski. West Pomeranian University of Technology, Poland</i>	
Automatic Synthesis of Computationally Efficient Interest Point Detectors	109
<i>Ivor Uhliarik. Comenius University, Slovakia</i>	
Automatic Detection of Shadow Acne and Peter Panning Artefacts in Computer Games	117
<i>Rafał Piórkowski. West Pomeranian University of Technology, Poland</i>	

3D Reconstruction

Adapting Hair and Face Geometry of Virtual Avatars with the Kinect Sensor	127
<i>Hannes Plank. Graz University of Technology, Austria</i>	
Improved 3D Reconstruction using Combined Weighting Strategies	135
<i>Patrick Stotko. University of Bonn, Germany</i>	
3D reconstruction of buildings from LiDAR data	143
<i>Marko Bizjak. University of Maribor, Slovenia</i>	

Sponsors of CESCg 2015

Invited Talks

Computational Challenges in Designing Virtual Models for Fabrication

Bernd Bickel

Institute of Science and Technology
Austria

Abstract

3D printing is considered a disruptive technology with potentially tremendous socioeconomic impact. In recent years, additive manufacturing technologies have made significant progress in terms of both sophistication and price; they have advanced to a point where devices now feature high-resolution, full-color, and multi-material printing. Nonetheless, they remain of limited use, given the lack of efficient algorithms and intuitive tools that can be used to design and model 3D-printable content. My vision is to unleash the full potential of 3D printing technology with the help of computational methods. In our research, we are working to invent and develop new computational techniques for intuitively designing virtual 3D models and bringing them to the real world. Given the digital nature of the process, three factors play a central role: computational models and efficient representations that facilitate intuitive design, accurate and fast simulation techniques, and intuitive authoring tools for physically realizable objects and materials. In this talk, I will present several projects that demonstrate our recent efforts in working toward this goal, structured according to basic object properties, and the lessons learned from working over several years with various 3D printers.

Bibliographical Details

Bernd Bickel joined IST Austria in early 2015 as Assistant Professor. He is a computer scientist interested in computer graphics and its overlap into animation, biomechanics, material science, and digital fabrication. His main objective is to push the boundaries of how digital content can be efficiently created, simulated, and reproduced.

Bernd obtained his Master's degree in Computer Science from ETH Zurich in 2006. For his PhD studies, Bernd joined the group of Markus Gross who is a full professor of Computer Science at ETH Zurich and the director of Disney Research Zurich. From 2011-2012, Bernd was a visiting professor at TU Berlin, and in 2012 he became a research scientist and research group leader at Disney Research, where he investigates approaches for simulating, designing, and fabricating materials and 3D objects.

Bernd's work focuses on two closely related challenges: (1) developing novel modeling and simulation methods, and (2) investigating efficient representation and editing algorithms for materials and functional objects. Recent work includes: theoretical foundations and practical algorithms for measuring and modeling the deformation behavior of soft tissue; simulating and reproducing fundamental properties, such as elasticity, surface reflectance, and subsurface scattering; and computational design systems for efficiently creating functional artifacts such as deformable objects and mechanical systems.

Challenges in Digital Shape Reconstruction

Tamás Várady

Budapest University of Technology and Economics
Hungary

Abstract

Digital Shape Reconstruction is a rapidly expanding new technology to produce complex digital models from measured data. A wide variety of applications emerges in mechanical engineering, medical sciences, and preserving the cultural heritage of mankind.

This talk focuses on reproducing engineering objects and investigates the limits of creating perfect CAD models in a semi-automatic manner. It gives an overview of the whole DSR process from 3D data acquisition through algorithms to reproduce the original design intent to 3D printing. Intermediate geometric processing steps include merging point clouds, repairing and simplifying meshes, segmentation, classifying surface regions, fitting surfaces and perfecting the final models before these are exported into CAD/CAM systems. Some interesting algorithmic details will also be discussed. The talk will be supplemented with lots of short videos that will help to gain some practical insight into this highly complex, interdisciplinary area.

Bibliographical Details

Dr. Tamás Várady led the Geometric Modeling Laboratory at the Computer and Automation Institute of the Hungarian Academy of Sciences for 20 years (1984 - 2003). He acted as Chief Technology Advisor for Geomagic, Inc., USA, and partly directed the development of their flagship product, Geomagic Studio (2003 - 2010). Currently he is a full professor at the Budapest University of Technology and Economics; his research interest includes digital shape reconstruction and surface modeling based on general topology curve networks. Dr. Várady has published more than 90 research papers, and his work is highly cited (2500+). He is an associate editor of Computer-Aided Geometric Design (Elsevier).

GPU

Accelerated gSLIC for Superpixel Generation used in Object Segmentation

Robert Birkus*

Supervised by: Ing. Wanda Benesova, PhD.[†]

Institute of Applied Informatics
Faculty of Informatics and Information Technologies STU
Bratislava

Abstract

The goal of our work is to create a robust object segmentation method which is based on superpixels and will be able to run in real-time applications.

The SLIC algorithm proposed by Achanta et al. [2] is a superpixel segmentation algorithm based on k-means clustering, which efficiently generates superpixels. It seems to be a good trade-off between the time consumption and robustness. Important advancement towards the real time applications using superpixels has been proposed by the authors of the gSLIC - a modified SLIC implementation on the GPU (Graphics Processing Unit) [11].

In this paper, we present a significant acceleration of this superpixel segmentation algorithm gSLIC implemented for the GPU. A different strategy of the implementation on the GPU speeds up the calculation time twice and more over the presented GPU implementation. This implementation can work in real-time even for high resolution images. We also present our method for merging of similar superpixels. This method uses an adaptive decision procedure for merging of superpixels. Accelerated gSLIC is the first part of this proposed object segmentation method.

Keywords: Superpixel, Image segmentation, GPU, SLIC, gSLIC, Region merging, Real-time

1 Introduction

Superpixels are produced by a deliberate oversegmentation of an image with the goal to generate segments which can serve as basic units in the further image processing. Superpixels are able to increase calculation efficiency viewed from the next processing task because of the reduction of the redundancy in the image. Superpixels are expected to be regular sized as often as possible but on the other hand, they should follow the saliency edges in the image. To distinguish between a high salience and a low salience edge is quite a complicated task, mainly if the

processing is performed on a small local area in the image. Hence, the development of a robust superpixel segmentation algorithm, which could run in the real-time applications, remains a challenge in the computer vision tasks. In many areas of application, the effort involved in computing in real-time is of high importance.

Typical real-time applications are from the area of video processing using superpixels, where several approaches could be combined with frame-based superpixel segmentation as, for example, the approach which has been presented in the paper by Chang et al. [3]. A reduction of the time consumption in the superpixel segmentation is needed also in applications based on still images, especially if we need to process large images.

One of the further image processing tasks after superpixel segmentation is the object segmentation. Automatic universal object segmentation is one of the most common problems in computer vision. To achieve object segmentation using the superpixels, superpixels, which belong to the same object must be merged. This seems to be a simple task, but in fact it is a big challenge. The real difficulties are already open in the implementation. The basic idea is to merge similar superpixels together, but the real question is: which superpixels are similar? There are lots of features and variables to consider.

In this paper we firstly summarize some selected already published methods of superpixel segmentation. We focus on the SLIC algorithm. Further, we also summarize some GPU implementations of superpixel segmentation algorithms. The focus is predominantly on the parallel implementation of the SLIC algorithm on GPU, called gSLIC. Our main contribution is the acceleration of the gSLIC implementation using parallel reduction. We describe our implementation in detail and discuss different key points of our implementation, specially those which are important from the point of view of efficiency. We also describe our CPU implementation of our superpixels merging algorithm, which merges the superpixels based on an adaptive threshold according to the color difference of the most similar neighbor. We also show some results of the merging procedure. The paper contains detailed results of the segmentation quality of SLIC and gSLIC and

*rbirkus@gmail.com

[†]vanda.benesova@stuba.sk

also the speedups of our implementation compared to the gSLIC implementation. At the end of this paper we briefly discuss our achieved results and present our future work.

2 Related work

Selected already published methods of superpixel segmentation and region merging are briefly summarized in this section.

2.1 Superpixel segmentation

In Spatially Coherent Clustering using Graph Cuts [14], Zabih and Kolmogorov propose a method with the goal to overcome the absence of spatial coherence in segmentation while a clustering in feature space is used. An energy function which consists of a term representing the energy in the spatial space and a term representing the energy in the feature space has to be minimized using graph cuts.

Veksler and Boykov [13] formulate the superpixel partitioning problem in an energy minimization framework, and optimize with graph cuts. The presented energy function explicitly encourages regular superpixels and this method is also suitable for 3D supervoxel segmentation. An image is covered with overlapping square patches of fixed size. Hence, each pixel is covered by several patches, and the task is to assign a pixel to one of them. TurboPixels [8] is an iterative algorithm which starts by evolution from seeds placed regularly in the image. The algorithm then iterates until no further evolution is possible, i.e., when the speed at all boundary pixels is close to zero. The iteration loop involves: an evolution of this boundary, estimation of the skeleton of the unassigned region and updating of the speed of each pixel on the boundary and of unassigned pixels in the immediate vicinity of the boundary.

Shi and Malik [12] propose a graph-theoretic criterion for measuring the goodness of an image partition - the normalized cut. The authors showed that the minimization of this criterion can be formulated as a generalized eigenvalue problem. A computational method based on this idea has been developed and presented by the authors and applied to segmentation of brightness, color, and texture images.

Felzenszwalb and Huttenlocher [5] define a predicate for evaluating two regions of an image whether or not there is evidence for a boundary between two components in a segmentation. This predicate is based on measuring the dissimilarity between elements along the boundary of the two components relative to a measure of the dissimilarity among neighboring elements within each of the two components.

2.1.1 SLIC algorithm

R. Achanta et al. introduced the Simple Linear Iterative Clustering (SLIC) [1] method for producing compact and nearly uniform superpixels. The efficiency, simplicity and the performance of the algorithm makes it widely used and often modified with the goal to achieve even better performance. Yuheng Ren and Ian Reid introduced a parallel implementation of the SLIC superpixel segmentation [11], called gSLIC. The gSLIC implementation uses GPU and the NVIDIA CUDA framework and is able to achieve speedups of about 10x to 20x over the native SLIC sequential implementation.

The SLIC algorithm is based on the k-means clustering principles. Each pixel is associated with a 5-dimensional feature vector $[L*a*b \ x \ y]$, where $L*a*b$ are color coordinates in CIE $L*a*b$ space and x, y are spatial coordinates. $L*a*b$ color space was designed, so that color differences measured as Euclidean distance in the $L*a*b$ space correspond with color differences given by human perception. Although the used conversion from RGB to $L*a*b$ includes conversion errors due to missing information about the spectral characteristics of the used camera, this error seems to be irrelevant and using $L*a*b$ coordinates better results could be achieved than with using RGB color coordinates. In the initialization step, positions of all seeds are defined in a regular grid step S , up to a small shift to avoid image edges. The grid size is calculated as in Equation 1, where N is the number of pixels in the given image and k is the required number of segments.

$$S = \sqrt{\frac{N}{k}} \quad (1)$$

The rough size of each superpixel is then given as a square of the grid size S . After the mentioned initialization step the k-means clustering will be calculated for each seed and subsequently the position of the seed will be iteratively updated - shifted into the center of the newly derived cluster. More iterations (typically 5 to 20) are necessary for the useful segmentation result. Finally, in the last step named enforce connectivity, extremely small superpixels will be removed - included within another superpixel. For a balance between a regular form of the superpixels and a form of superpixels given by the color differences, a compactness constant has to be defined and used in the distance measure definition as a weighting factor.

2.2 Superpixels - GPU implementations

Brian Fulkerson and Stefano Soatto introduced a parallel GPU implementation of Quick Shift: they called it Really Quick Shift [6]. This implementation is able to achieve speeding up of 10x to 50x over the original CPU version of Quick Shift. Quick Shift operates on each pixel in the image independently of its distant surroundings. Hence, the GPU implementation basically follows the steps of native

Quick Shift because the Quick Shift is a good parallelizable algorithm. Fulkerson et al. first built a simple GPU version of Quick Shift that simply copies the image to the device and breaks the computation of the density and the neighbors into blocks for the GPU to process. This simple GPU version is faster than the CPU version. However, Quick Shift needs many memory accesses and the global memory of the GPU is slow. Memory latency is a bottleneck for this GPU version. To avoid memory latency they used the advantage of GPUs shared memory. They load an apron of pixels surrounding the block into shared memory. However, this operation is not easily separable and the shared memory is quickly exhausted. So, instead of this solution they map the image and the estimate of the density to a 3D and 2D texture. This GPU version is faster than the previous one. They evaluated both GPU implementations and the CPU implementation as well. This GPU implementation of Quick Shift provides a 10 to 50 times speedup over the CPU implementation, resulting in a super-pixelization algorithm which can run at 10Hz on 256x256 images.

2.2.1 gSLIC implementation

Parallel implementation of the Simple Linear Iterative Clustering (SLIC) superpixel segmentation (gSLIC) is proposed by Carl Yuheng Ren and Ian Reid [11]. The implementation was applied and tested using GPU and NVIDIA CUDA framework. The authors have presented speedups of 10x to 20x on a single graphics card compared with the CPU sequential implementation. The presented gSLIC algorithm differs from the originally proposed SLIC algorithm in the way in which the clustering is carried out. Whereas the SLIC algorithm uses a clustering procedure based on an evaluation in the surrounding of each seed in the $2S \times 2S$ region of pixels, gSLIC algorithm is running in the opposite way. Each pixel is associated with the 9 nearest cluster centers and the search is running for the nearest of the nearby 9 cluster centers. Therefore, the pixel will be labeled with the nearest cluster's index.

Hence, gSLIC has been modified in order to carry out the reasonable part of the parallel computing on GPU by one-thread-per-pixel computing. In general, gSLIC can be split into two parts: CPU and GPU. The image has to be acquired by the host function running on the CPU, then it can be transferred to the GPU device memory. Henceforth, the main part of the algorithm: color space transformation (RGB to L^*a^*b) and clustering will be carried out on the GPU. Subsequently the derived segmentation mask will be transferred back to the host function again, where a recursion-based post processing function runs to enforce the connectivity of all superpixels.

The color space transformation part is naturally pixel-wise parallelizable, so gSLIC uses one thread per pixel on 16×16 blocks. Then one thread per cluster will be used to initialize cluster centers. Next in the assignment step each thread takes care of one pixel. However, the block

assignment is a little more complex. The initial size of each cluster is determined by S , where S is the grid interval calculated as in Equation 1. In most cases, the size of each cluster is larger than the thread block size, thus clusters consist of multiple thread blocks. This block assignment guarantees that all threads of the thread block need to search the same set of cluster centers in the neighborhood for the nearest one. Thus gSLIC pre-load the cluster centers' information into local shared memory for higher efficiency. In each iteration after all pixels have been assigned with a label (which is the index of the nearest center), gSLIC uses one thread per cluster to update the cluster center. After the k-means iteration has converged, the labeled image will be transferred back to the host as a segmentation mask. The post-processing to enforce connectivity is the same implementation as in SLIC.

2.3 Region merging

J. Ning et al. [10] present a region merging based interactive image segmentation method. The image is initially segmented by mean shift segmentation and the users only need to roughly indicate the main features of the object and background by using some strokes, which are called markers. With the similarity-based merging rule, a two-stage iterative merging algorithm was presented in the paper [10] to gradually label each non-marker region as either object or background.

H. Dunlop et al. [4] propose a detection and segmentation method incorporating features from multiple scales. This method was tested for the identification of rock appearances and has been evaluated on representative images from the Mars Exploration Rover catalog. This method uses a superpixel segmentation followed by region-merging to search for the most probable groups of superpixels. The authors believe that the method provides promising results for object identification in natural scenes.

3 Our contributions

In this section we summarize all of our contributions in detail.

3.1 Accelerated gSLIC

Our main contribution is a different strategy of the implementation which has been done in the cluster centers updating part. It is also mentioned by the authors of gSLIC [11] that this part could be accelerated by using a parallel reduction algorithm. Based on the technical report by Mark Harris [7] using all of the optimization techniques a significant improvement of the time consumption has been achieved. We also optimized the color space conversion (RGB to Lab) using floating-point constants instead

of double-precision constants. This optimization eliminates the redundant conversions from double to float.¹

The principle of our implementation does not differ from the principle of gSLIC. The only difference is in the work management of threads. To calculate the new cluster center gSLIC searches over a $2S \times 2S$ region around the current cluster center with one thread. The reason why the authors of gSLIC used only one thread to calculate the mean value of this region of pixels is to avoid atomic operation. To avoid both the atomic operation and using only one thread an implementation of parallel reduction is needed. In contrast to gSLIC, our implementation also works with the $2S \times 2S$ region of pixels around the current cluster center, but instead of one thread we are using a block of threads to do the mean value calculation.

In most cases the size of the $2S \times 2S$ region is larger than the size of thread block. To maximize the number of threads working on one region we could use more thread blocks than gSLIC did at the k-means iteration. However in parallel reduction the threads need to share some data. To efficiently share data between threads we have to use the shared memory of CUDA, even if we can use only one block of threads per region due to the shared memory's block restriction. We also considered using the global memory to share data between multiple blocks of threads, but it is much slower than the shared memory and even with L1 cache memory it cannot compete with the shared memory in this task. However, in case of extremely large regions, maybe the solution with multiple thread blocks using global memory would be better. We have not tested it yet.

In the first step of our implementation due to the larger region size as the block size each thread of the thread block calculates preliminary results from multiple pixels and stores it in the shared memory. When the amount of data is reduced to the number of threads per block the parallel reduction begins.

3.1.1 Occupancy

The occupancy can be defined as a proportion of active threads and maximum active threads. To hide the latency and gain maximum efficiency we must have as many active threads as possible. Let us consider Kepler GK104 architecture in the following calculations. Its limitations are shown in the Table 1.

In our implementation we need 24 bytes of shared memory per thread to store the temporary results. So, on GK104 architecture we can have $48kB/24B = 2048$ active threads per multiprocessor, which is the maximum. That means the shared memory usage does not degrade our efficiency. However, if we would need, we can decrease the usage of shared memory by using short int variables instead of int variables. We decided to set the block size to 128. This block size gives us occupancy equal to 1. If we

¹The source code will be available on the web site <http://vgg.fiit.stuba.sk/image-segmentation-on-gpu/>

	KEPLER GK104
Compute Capability	3.0
Threads / Warp	32
Max Warps / Multiprocessor	64
Max Threads / Multiprocessor	2048
Max Thread Blocks / Multiprocessor	16
32-bit Registers / Multiprocessor	65536
Max Registers / Thread	63
Max Threads / Thread Block	1024
Shared Memory Size (bytes)	48K

Table 1: Limitations of Kepler GK104 architecture

would choose a smaller block size, for example 64, due to the maximum number of blocks per multiprocessor (16) we could have only $64 * 16 = 1024$ active threads per multiprocessor. However, we could choose larger block sizes up to 1024 and the occupancy would be still equal to 1. The reason why we chose the block size 128 is explained in the next section. The most efficient block size for our implementation is architecture-dependent. The register usage does not affect the efficiency of our implementation because each thread can use up to $65536/2048 = 32$ registers without decreasing the occupancy and in our implementation each thread uses only 28 registers.

3.1.2 Parallel reduction optimization

After the amount of data is reduced to the number of threads per block we make a parallel reduction in shared memory. We are using almost all of the optimization techniques by Mark Harris [7]. We are avoiding using % operator wherever it is possible because it is very slow.

To achieve high memory bandwidth for concurrent accesses, shared memory is divided into equally sized memory modules, called banks, that can be accessed simultaneously. To avoid bank conflicts we are using sequential addressing instead of interleaved addressing. In Figure 1 we show the two types of addressing. Let us consider memory banks of size 4 bytes. To access four elements sequentially we need only one transaction because all of the accessed elements are in the same memory bank. However, using the interleaved addressing we need two transactions because those four accessed elements are situated in two different memory banks.

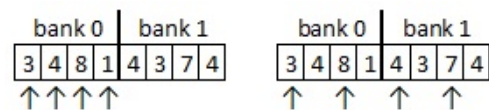


Figure 1: Sequential addressing (left) and interleaved addressing (right)

In parallel programming to achieve high efficiency we have to avoid idle threads as much as we can. Thankfully

for the cases in which the region size is larger than the block size, each thread has to handle multiple pixels and there are no idle threads. Mark Harris also mentioned in his technical report [7] based on Brent's theorem that it is beneficial for each thread to do more sequential work and this is the reason why we chose the smallest block size (128), which give us occupancy equal to 1. If we would choose larger block sizes then the sequential work for each thread would be less and there could be idle threads.

To save useless work in all threads of the thread block we unrolled the for loop by putting `#pragma unroll` directive right before the loop. `#pragma unroll` is a compiler optimization which unroll the loop. However, the number of iterations of the loop have to be known in compile time. In our case the number of iterations is determined by the block size, which is defined as a macro constant. Without unrolling the for loop, all threads would do additional operations every iteration of the loop.

3.2 Superpixel merging

Accelerated gSLIC is the first part of the object segmentation algorithm proposed as bottom-up segmentation using superpixels. The main idea can be briefly described as the merging of similar superpixels. Despite this quite simple and native approach, more challenges are already open in the implementation. The first one is a decision about the similarity of two superpixels. This is quite a complicated task which is a crucial part of our research. The second one is time optimization of the whole segmentation procedure using GPU.

In this paper we present the results of our CPU implementation of the presented merging algorithm using the metric:

$$\Delta D_{Lab} = \sqrt{(L_2^* - L_1^*)^2 + (a_2^* - a_1^*)^2 + (b_2^* - b_1^*)^2} \quad (2)$$

where (L_1^*, a_1^*, b_1^*) is the mean $L^*a^*b^*$ of the first superpixel and (L_2^*, a_2^*, b_2^*) is the mean $L^*a^*b^*$ of the second superpixel. The whole merging algorithm is presented in Figure 2.

Fast superpixel oversegmentation has been implemented by accelerated gSLIC as presented in the previous section.

The goal of the next section is to label all similar neighbor superpixels of each given superpixel. As mentioned, the decision about the similarity is a complicated task. We wanted to avoid a fixed threshold in the decision about the similarity because of the low invariance of such a threshold. Our decision is based on the relative threshold in relation to the superpixel whose distance ΔD_{Lab} is the least of all neighboring superpixels. The decision about the accepted similar superpixels is the following:

The neighboring superpixel will be labeled as similar if it satisfy the condition:

$$Dist < C * Dist_{min} \quad (3)$$

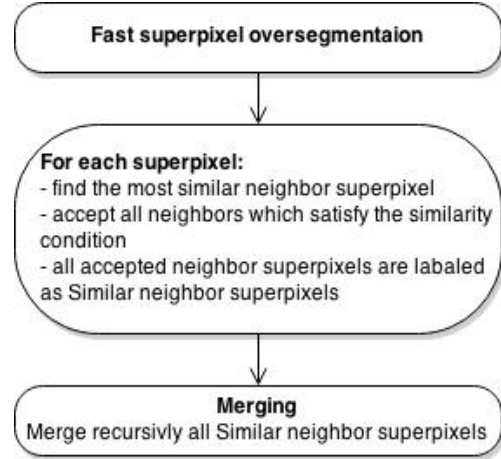


Figure 2: Merging algorithm

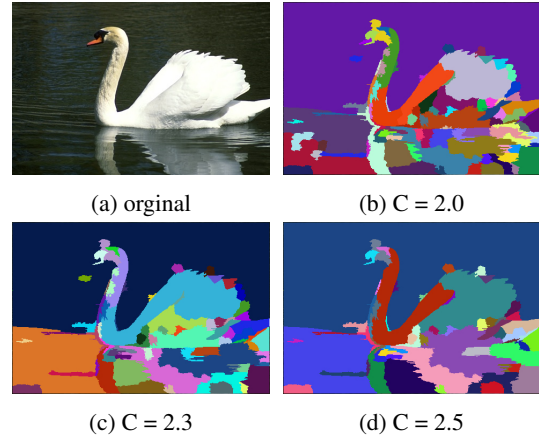


Figure 3: First example of merged superpixels

where $Dist$ is distance between the superpixels and C is a constant. Our results have been evaluated for $C = 2.0$, $C = 2.3$ and $C = 2.5$.

In last step "Merging", all superpixels labeled as similar will be recursively merged into one segment.

Examples of merged superpixels can be seen in Figure 3 and Figure 4. Merging using a higher constant C produces less new segments, but the undersegmentation error will be probably higher.

4 Results

The time needed for the superpixel calculation using modified gSLIC in comparison with the gSLIC [11] has been evaluated. Table 2 gives the profiling of the clustering part of the algorithm calculated on the NVIDIA GT 740m. The table gives the evaluation of the clustering part of the SLIC algorithm. Time profiling using the NVIDIA GTX 770 for the SLIC clustering is presented in the Table 3. The transfer times between host and GPU memory are not considered in the presented results.

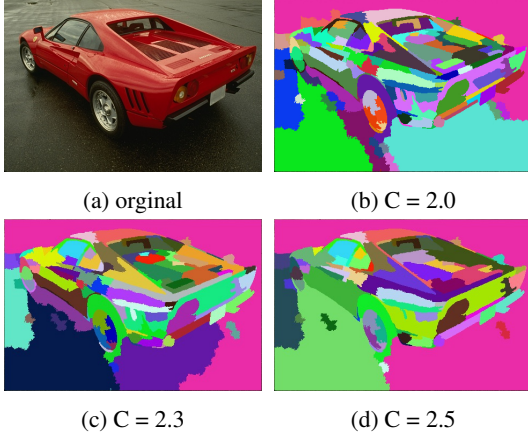


Figure 4: Second example of merged superpixels

Image size	gSLIC	Accelerated gSLIC		Speed-up
	CC 1.0	CC 1.0	CC 3.0	
320x240	7.06 [ms]	1.69[ms]	1.61 [ms]	4.18x
640x480	17.08 [ms]	6.72 [ms]	6.45 [ms]	2.54x
1280x960	66.16 [ms]	27.89 [ms]	26.31 [ms]	2.37x

Table 2: Time evaluation on the NVIDIA GT 740m.

As mentioned, SLIC and gSLIC clustering work differently. Actually the gSLIC is limited in the searching area of every pixel. Although this limitation is marginal, the question is, if this limitation has influence on the quality of the clustering. Therefore, we have evaluated the quality of the segmentation using SLIC and gSLIC and also the modification of the post-processing using the Superpixel Benchmark Toolbox [9] in order to achieve comparable results. The evaluation requires a ground truth segmentation made by humans, available in the dataset. Boundary recall (BR) is the fraction of hand-segmented edges which lie within a threshold distance k of any superpixel edge (in our experiments, $k = 2$). Since there can be multiple ground truth images for a single input image, they are added together using the OR operation.

The true positives (TP) count is the number of pixels in hand-segmented image, for which there is a superpixel boundary pixel in range k . The false negative (FN) count is the number of pixels in the hand-segmented image for which there is no superpixel boundary pixel in range k . Given these, we can calculate the boundary recall BR as in Equation 4:

$$BR = \frac{TP}{TP + FN} \quad (4)$$

The disadvantage of this metric is that it does not take into account the direction of the edges. Superpixel borders which intersect hand-segmented edges also contribute to the boundary recall. This metric also does not distinguish between superpixel edges which are off by 0, 1 and 2 pixels they all contribute to the boundary recall equivalently.

Results of the evaluation are presented in Figure 5. The

Image Size	gSLIC	Accelerated gSLIC	Speed-up
320x240	5.326 [ms]	0.426 [ms]	12.5x
640x480	8.0 [ms]	1.539 [ms]	5.20x
1280x960	19 [ms]	6 [ms]	3.17x
2560x1920	87.68 [ms]	24.12 [ms]	3.63x

Table 3: Time evaluation on the NVIDIA GTX 770 (using Compute Capability 1.0).

number of iterations was 5 and 10 and the compactness constant was set to 10 and 20. More detailed comparison for the nominal number of superpixels 150 is shown in Figure 6 and Figure 8. From Figure 5 it can be seen that with the rising number of segments the boundary recall results are better. We also can see that the increased number of iterations have not improved the quality of the segmentation that much. However, the compactness constant has a much bigger impact on the segmentation quality. Compactness constant 10 gives much better results than the compactness constant 20. From the evaluation of boundary recall on multiple images in Figure 6 we can see that results are unequivocal. In some of the images SLIC has better results and in others gSLIC has better results, but in average gSLIC gives us better results in boundary recall than SLIC.

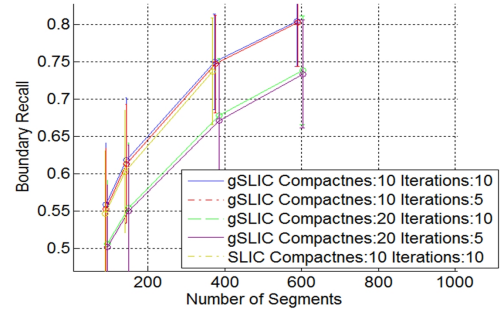


Figure 5: Boundary recall

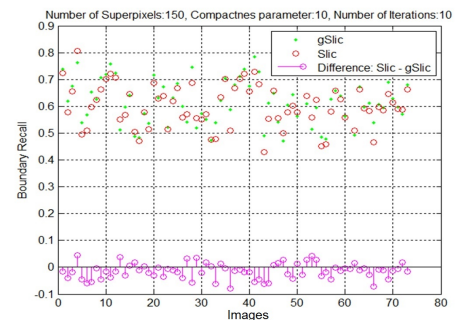


Figure 6: Comparison of the boundary recall for 73 images (No. of superpixels:150)

The undersegmentation error (UE) describes how much area of superpixels crosses the hand-segmented edges. Please refer to the original paper [9] for more information

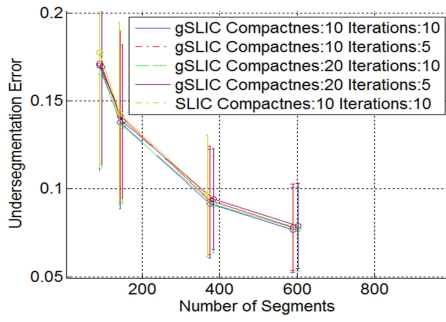


Figure 7: Undersegmentation error

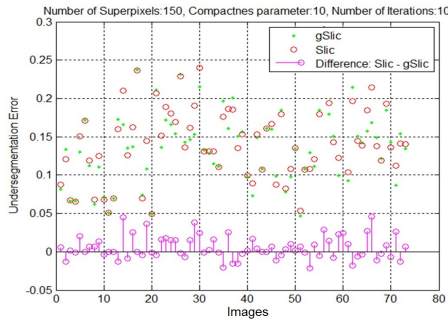


Figure 8: Comparison of the undersegmentation error for 73 images (No. of superpixels:150)

on its calculation. The evaluation of the undersegmentation error is presented in Figure 7 and Figure 8. From Figure 7 it can be seen that with the rising number of segments the undersegmentation error evaluation is better. We can also see that the different number of iterations or different compactness constants have very little impact on the results of the undersegmentation error evaluation. From the evaluation of undersegmentation error on multiple images in Figure 8 we can see that in some images SLIC is better and in others gSLIC is the better one. However, in average SLIC gives better results in undersegmentation error than gSLIC.

The goal of the merging of the superpixels is to reduce the number of segments while the boundary recall is high. The best possible value of boundary recall is the value at the original number of superpixels. In our case the original number of superpixels is 1027 as presented in the Figure 9. Boundary recall values of merged segments are remarkably better compared to the original SLIC segmentation.

5 Conclusions

We have presented modifications in the clustering part of the gSLIC algorithm. We implemented a parallel reduction and achieved significant acceleration as you can see above in Table 2 and Table 3. The modification does not have impact on the quality of the segmentation. We have also presented a comparison between SLIC and gSLIC.

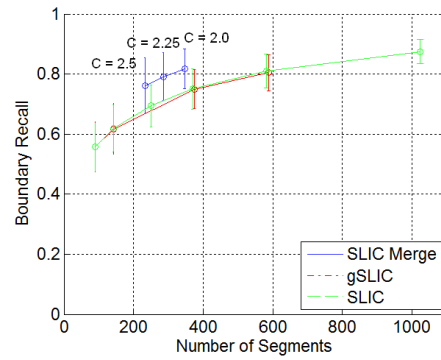


Figure 9: Boundary recall of the merged SLIC superpixels ($C = 2.0$, $C = 2.25$ and $C = 2.5$) in comparison with original SLIC and gSLIC

The results of the comparison is in average the same, depending on the evaluation method. SLIC has better results in undersegmentation error, in Boundary recall gSLIC is the better one. Finally, we have also presented in this paper a bottom-up method of segmentation using superpixels. The achieved results in boundary recall values are better than with using the original SLIC segmentation.

6 Future work

The gSLIC image segmentation algorithm consists of two parts. We successfully accelerate the clustering part, but the "enforce label connectivity" part is hardly parallelizable because it is a recursive function. We tried a completely different strategy based on the morphological processing. We achieved some small acceleration, but it was at the cost of segmentation quality. In our future work we would like to continue in this task to achieve acceleration without any quality degradation.

We presented an algorithm of merging superpixels. In the future we would like to accelerate the presented algorithm using the GPU implementation.

Our future work also contains research of features and decision-making procedures about the similarity of two superpixels. Our next experiments will include texture description and advanced classifications.

7 Acknowledgments

This research has been supported by a grant VEGA 1/0625/14.

References

- [1] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Süsstrunk. Slic superpixels. Technical report, Ecole Polytechnique Fedrale de Lausanne, Report No. EPFL-REPORT-149300, 2010.

- [2] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Susstrunk. SLIC Superpixels. Technical report, EPFL, 2010.
- [3] Jason Chang, Donglai Wei, Fisher III, and John W. A Video Representation Using Temporal Superpixels. *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pages 2051–2058, June 2013.
- [4] Heather Dunlop, David R Thompson, and David Wettergreen. Multi-scale features for detection and segmentation of rocks in mars images. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–7. IEEE, 2007.
- [5] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient graph-based image segmentation. *Int. J. Comput. Vision*, 59(2):167–181, September 2004.
- [6] Brian Fulkerson and Stefano Soatto. Really quick shift: Image segmentation on a gpu. Technical report, Department of Computer Science, University of California, Los Angeles, 2010.
- [7] Mark Harris. Optimizing Parallel Reduction in CUDA. Technical report, nVidia, 2008.
- [8] Alex Levinstein, Adrian Stere, Kiriakos N. Kutulakos, David J. Fleet, Sven J. Dickinson, and Kaleem Siddiqi. Turbopixels: Fast superpixels using geometric flows. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(12):2290–2297, 2009.
- [9] P. Neubert and P. Protzel. Superpixel benchmark and comparison. In *Proc. of Forum Bildverarbeitung*, 2012. Regensburg, Germany.
- [10] Jifeng Ning, Lei Zhang, David Zhang, and Chengke Wu. Interactive image segmentation by maximal similarity based region merging. *Pattern Recognition*, 43(2):445–456, 2010.
- [11] Carl Yuheng Ren and Ian Reid. gSLIC: a real-time implementation of SLIC superpixel segmentation. Technical report, University of Oxford, Department of Engineering, Technical Report (2011)., 2011.
- [12] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, CVPR '97, pages 731–, Washington, DC, USA, 1997. IEEE Computer Society.
- [13] Olga Veksler, Yuri Boykov, and Paria Mehrani. Superpixels and supervoxels in an energy optimization framework. In *Proceedings of the 11th European conference on Computer vision: Part V, ECCV'10*, pages 211–224, Berlin, Heidelberg, 2010. Springer-Verlag.
- [14] Ramin Zabih and Vladimir Kolmogorov. Spatially coherent clustering using graph cuts. In *CVPR (2)*, pages 437–444, 2004.

Real-time Water Simulation with Wave Particles on the GPU

Daniel Mikeš*

Supervised by: Jiří Bittner

Department of Computer Graphics and Interaction
Czech Technical University in Prague, Czech Republic

Abstract

When rendering large bodies of water in real-time an efficient method is required to model water waves. This article describes a method for real-time interactive generation of such waves. We use the wave particle method to describe wave propagation in a fluid medium. The method allows to simulate interactions of water with general shaped rigid bodies in real-time. We present a GPU implementation of the method and show results in scenarios such as open ocean waters or pools with water boundaries.

Keywords: Wave particle, GPU algorithm, interactive, realtime simulation, waves, water rendering

1 Introduction

In real-time applications such as video games it is crucial that all the computations are fast enough to be computed in a plausible frame rate, so both the rendering and the simulation stages should be fast. In general lots of computationally complex tasks can be measured or pre-computed and then used later on. On the other hand user interaction cannot be simply predicted so the computation must run on-line. For these reasons we usually need to settle for approximate solutions meaning the rendering and the simulation step is not necessarily physically correct but offers reasonable results.

Animation of large bodies of water such as lakes or oceans is important part of computer graphics and it is still an open challenge since 3D volumetric simulations are too computationally complex for mentioned scenarios.

In this article, we focus on a real-time water simulation of large bodies of water with local surface waves. We address height field representation to describe the water surface. In our simulation we use *wave particles* as a spatial information about the water surface deformation. The simulation utilizes the graphic hardware to efficiently distribute the wave particles onto the water plane according to the motion of the rigid body in the water medium.

This article is based on the work of Yuksel et al. [12]. They presented the original wave particles method which

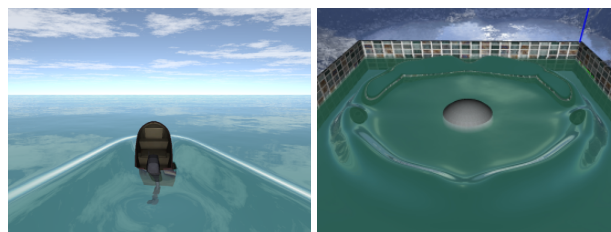


Figure 1: Real-time water simulation running on the GPU. Open ocean scenario with boat interaction (left, 290 FPS), Pool scenario with falling object (right, 270 FPS).

is briefly described in section 3. In contrast to their work, we address using wave particles specifically on the GPU.

2 Related Work

Raveendran et al. [10] proposed a method for preserving uniform particle density in SPH. Onderik et al. [8] presented a SPH method improvement with small scale details such as splashes and foam. A method which uses Eulerian approach to simulate 3D water volume with a grid cell reduction was described by Irving et al. [7].

Chen et al. [2] used height field representation with spatial domain waves using shaders and bump mapping to create small ripples on the water surface.

Chou et al. [3] described a simple method for ocean simulation with one-way interaction between the water surface and rigid bodies.

Galin et al. [5] presented a real-time interactive water simulation method. They address special type of waves created by the engine of the boat, which also creates foam on the water surface. Therefore this method is usable only for specific type of rigid bodies. In contrast to the wave particle method they use a 2D grid to represent the waves instead of particles, therefore the performance is dependent only on the grid size and independent of the number of wave fronts. They are also able to handle the diffraction effect but the overall performance is lower, compared to the wave particle method.

*dm.mikes@gmail.com

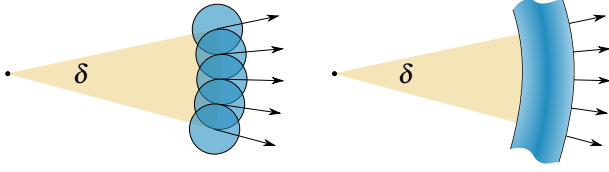


Figure 2: Wave front generation from particles. Source particles (left) and generated wave front (right) from the top view. Dispersion angle δ of the parent particle is depicted yellow.

3 Wave Particles

The wave particle method uses a particle system for representing surface deviation. Position \mathbf{x} of each particle is used for localizing the deviation function $d_v(\mathbf{x}, t)$ and they are totally independent of each other. Unlike Lagrangian methods wave particles move in a plane which is coplanar with the water surface. The wave particles do not represent elements of the water mass, but only a deformation on water surface.

Set of local deviation function is synthesized to the global deviation function

$$D_v(\mathbf{x}, t) = y_0 + \sum_{i \in P} d_{v_i}(\mathbf{x}, t), \quad (1)$$

where \mathbf{x} is the position on the water surface, t is the time, $d_{v_i}(\mathbf{x}, t)$ is the local vertical deviation function of the i -th particle, y_0 is water base level, and P is a set of all particles.

The local deviation function can be expressed as

$$d_{v_i}(\mathbf{x}, t) = \frac{A_i}{2} \left(\cos \left(\frac{\pi |\mathbf{x} - \mathbf{x}_i(t)|}{r_i} \right) + 1 \right) \Pi \left(\frac{|\mathbf{x} - \mathbf{x}_i(t)|}{2r_i} \right), \quad (2)$$

where A_i is the amplitude of i -th particle, r_i is the wave particle radius, \mathbf{x} represents a point of the water surface, $\mathbf{x}_i(t)$ is the position of the i -th wave particle in time t and Π is a box function, which limits cosine function over a finite region in 3D domain.

$$\Pi(x) = \begin{cases} 1, & -\frac{1}{2} \leq x \leq \frac{1}{2} \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

3.1 Longitudinal Waves

The motion of the water surface is not limited only to the vertical deviation. In reality water particles propagate in circles which creates sharper peaks on the surface waves as shown in figure 3.

$$d_{h_i}(\mathbf{x}, t) = d_{v_i}(\mathbf{x}, t) \left(-\mathbf{v}_i \sin \left(\frac{\pi \mathbf{u}}{r_i} \right) \Pi \left(\frac{u}{2r_i} \right) \right), \quad (4)$$

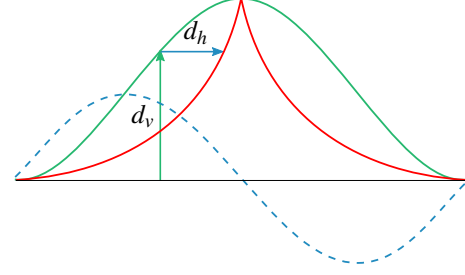


Figure 3: The horizontal (d_h) and vertical (d_v) deviation function. Original wave (green), the final wave (red), and the vertical deviation (dashed blue).

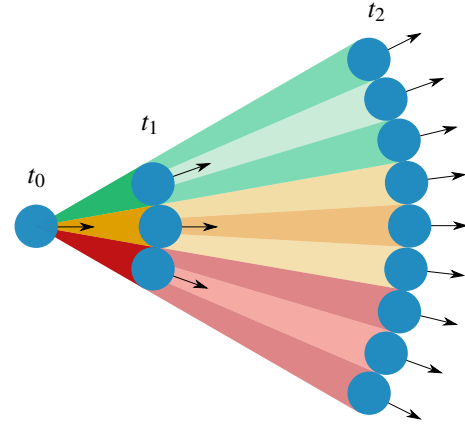


Figure 4: Dispersion angle partitioning after particle subdivision operation at different time steps (t_0 , t_1 , and t_2). Dispersion angle of each particle (blue circle with an identifier) is marked by different colour to enhance clarity.

where $\mathbf{u} = |\mathbf{x} - \mathbf{x}_i(t)|$, the propagation direction \mathbf{v}_i and Π is the box function. This model can introduce undesirable self intersections if $r_i A > 1$. The problem is addressed by a new parameter which affects the strength of longitudinal wave.

3.2 Particle Subdivision

An expanding wave front arises when a wave is created in one point and distributes further in all directions in a 3D domain as shown in figure 2. The local deviation function does not allow changing the size of the wave particle in time. This causes that wave particles are getting further from each other while the wave front propagates. We need to cover the whole size of propagating wave front by placing one particle next to each other, so that even with a constant particle size the wave front is continuous. This is achieved by particle subdivision routine shown in figure 4.

If the distance between two neighbouring particles is larger than a defined threshold, a new particle is created on each side of the *parent* wave particle. The amplitude of the parent particle is distributed to the *child* particles in order to conserve energy.

Since the particle velocity is constant we can compute in advance at which time the distance between neighbouring particles will be beyond this threshold:

$$w_t = w_0 + \delta \mathbf{v} |t - t_0|, \quad (5)$$

where δ is the dispersion angle, w_0 is the distance between neighbouring particles in the current time t_0 , and \mathbf{v} is the particle velocity.

The threshold is set proportionally to the particle radius r_i and it ensures that adjacent particles will never be further from each other than the threshold parameter.

3.3 Wave Particle Properties

Besides the actual position \mathbf{x} , each wave particle stores the propagation angle α , dispersion angle δ , origin \mathbf{o} and the amplitude A .

The propagation angle represents the wave particle direction in the 2D plane, dispersion angle δ is introduced to describe a spatial range in which new particles appear after subdivision process. The wave particle origin is the position of the particle at time $t = 0$ and it is fixed as the wave particle propagates. Amplitude represent the energy of the wave particle. Particles with low amplitude have also low contribution to the deviation function. In some scenarios it is also useful to model waves with negative amplitudes.

3.4 Creating new Wave Particles

After particle is subdivided two new particles are added to the system. Important property of the convincing physical simulation is the energy conservation criterion. Therefore the amplitude is evenly distributed to the newly created particles. The dispersion angle also changes because each particle now describes one third of the original range. Children particles are placed in the same distance r_δ from the origin. Descendants also inherit the origin of the parent particle.

3.5 Water Boundary

Scenarios such as pools require a model for reflecting incoming wave fronts off of the water boundary. The boundary represents the container which holds the simulated water.

The distance from the particle to the origin r_δ is important for the particle subdivision since it tells us when subdivision occurs. To handle particle reflection, the origin has to be mirrored over the boundary in order to persist the subdivision criterion. Since the distance between adjacent particles w does not change during the reflection, the dispersion angle may change owing to the curvature of the boundary as shown in figure 5.

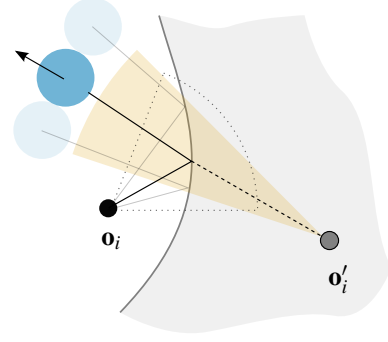


Figure 5: Wave particle (blue circle) reflection off of a curved boundary with origin \mathbf{o}_i . The grey area represents the boundary with the new mirrored origin \mathbf{o}'_i . The dotted sector represents the dispersion angle before reflection and the yellow sector is the dispersion angle after reflection.

$$w' = w \quad (6)$$

$$\delta' = \delta \frac{r_\delta}{r'_\delta}, \quad (7)$$

where δ is the dispersion angle immediately before reflection and δ' is the dispersion angle immediately after reflection; the same notation is used for other symbols. The curvature influences δ' indirectly via the origin \mathbf{o}' .

4 Wave Particles on the GPU

In our particle system it is essential to preserve the data on GPU memory without unnecessary data transfers from main memory to GPU buffer and vice versa. We use *attribute data* with *point* geometry to represent particles in OpenGL.

OpenGL Transform Feedback Buffer (TFB) allows us to capture the output of vertex or geometry shader inside the GPU memory. Location of TFB in the OpenGL pipeline is shown in figure 6. In each draw step the GPU fetches vertices¹ and pushes them in the vertex shader, where attribute properties can be modified or simply passed further in the pipeline. After that, the points can be stored in the TFB meaning that the data are persistent in one draw step.

The actual buffer where the primitives are stored can have different types. We use Vertex Buffer Object (VBO) as the destination of Transform Feedback operation because we reuse captured vertices in the next frame.

Therefore, we use two Transform Feedback Buffers and we chain them together in a way that output of the first buffer is the input of the second buffer as shown in figure 7. The TFBs are connected in the other way respectively. Two buffers are used due to the fact that OpenGL does

¹Point is a 1D primitive and can be represented by one vertex. Therefore, terms point, vertex and particle are interchangeable in this context.

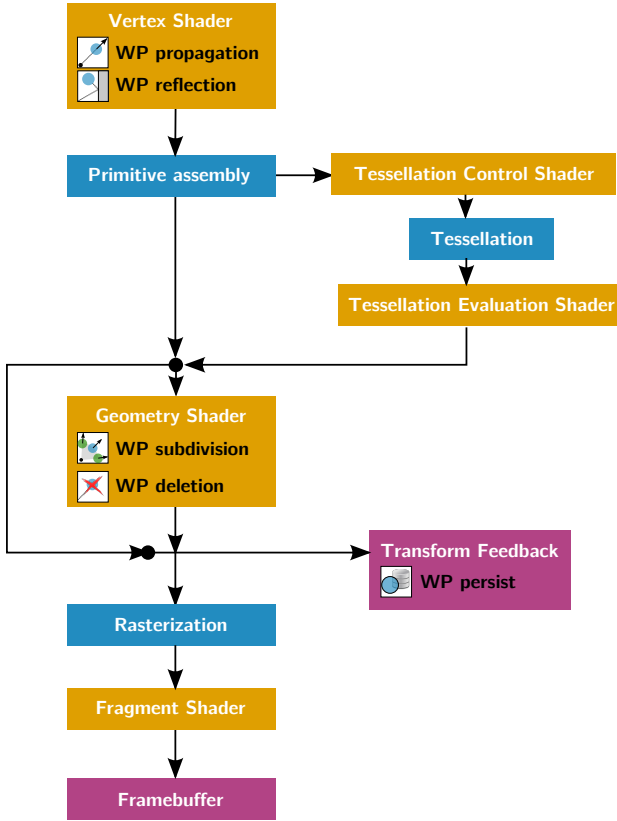


Figure 6: Simplified OpenGL pipeline. Yellow coloured boxes represents the programmable stages, blue are the fixed stages, and violet represents output buffers.

not allow reading from and writing into the same Vertex Buffer Object.

Wave particle propagation is done on the vertex shader. Only limitation in the vertex shader stage is that there is only one vertex in the input and one vertex on the output for one shader invocation. That means we cannot use vertex shaders for particle subdivision. For this purpose we can use tessellation shader or geometry shader, which is able to emit new vertices. Newly created vertices are then also stored in TFB. Figure 6 also shows which of the main tasks are performed in current stage of the OpenGL pipeline.

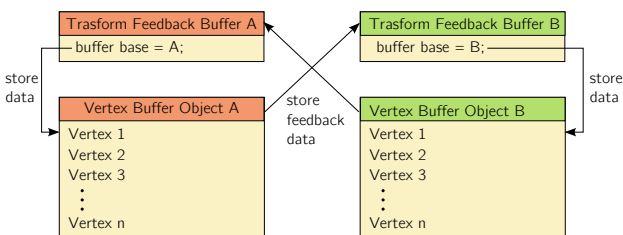


Figure 7: Illustration of Transform feedback buffer swapping. The output of first TFB is used as an input of the second TFB.

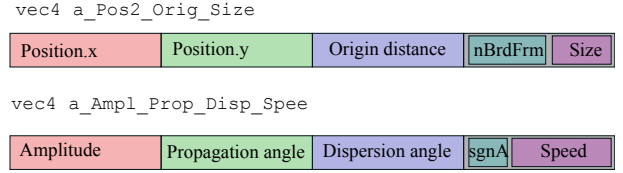


Figure 8: Wave particle structure encoded for the use on the GPU. Each field represents one floating point number.

4.1 Particle Data Structure

In order to avoid large memory consumption, we need to efficiently represent vertex attribute data.

Vertex attributes are internally packed and aligned into a multiple of vec4^2 in OpenGL[1]. This means that it is beneficial to use vec4 data type and fit all necessary information to it as few member variables as possible.

Figure 8 shows the GPU packing of wave particle structure.³ The wave particle structure is packed into 32 bytes.

Propagation routine The entry point for the simulation process is in the vertex shader, where particles are properly moved according to the propagation angle. If the particle should be subdivided or discarded, respective flag is set and the particle is passed further into the pipeline.

Subdivision and deletion routine Vertices are passed to the geometry shader, where the vertex can be either discarded or emitted. The particle is discarded if the amplitude is lower than a defined threshold and the influence of the wave particle is neglectable.

Particle generation routine Particle generation is the process of creating new particles based on the object to water interaction. The result of the interaction step is stored in the wave particle distribution texture. Wave particle distribution texture obtains information about spatial distribution of direct and indirect wave effect which is converted into particles in this step. Propagation direction is also part of the texture. The process of obtaining distribution texture is described further.

Each texel represents a potential wave particle. Therefore, we create a vertex buffer and fill it with vertices organized into 2D grid with the same resolution as the texture. Consequently we set the wave particles properties to the vertices from the texels and we convert the wave effect to the amplitude. Particles with non-zero amplitude are emitted and eventually captured by the TFB.

Wave particle reflection For performance purposes we represent boundaries as a texture. Normal vector of the boundary is encoded into each texel in order to compute

²GLSL representation of 4-dimensional 32-bit floating point number.

³nBrdFrames refers to number of consecutive frames behind border and is used for error correction in wave particle reflection routine.

the reflection. Discrete step collision detection can produce errors when the object is moving too fast and the object passes through the boundary in one frame. We adjust the texture mapping with respect to the particle speed to handle these situations.

4.2 Particle Filtering

Particles are rendered as circles with the radius equal to particle size. The final deviation of the water surface can be obtained by rendering all the particles with additive blending from a top orthographic view similarly to the texture splatting. Instead we use smaller points to represent the information about the particle presence and render them into texture. The wave particle render texture is filtered and the contribution of each wave particle in a local distance is accumulated. This is similar to the texture gathering process.

In this step wave particle deviation function is applied. The contribution of each wave particle in the filtering step is weighted by the deviation function in equation 2. Note that the function can be converted into separable filter. This means we can perform 1D filtering process consecutively for each axis and compose the final result.

The filter function can be denoted as

$$d_h^X(p) = \frac{1}{2} \left(\cos\left(\frac{\pi p}{r}\right) + 1 \right), \quad (8)$$

$$d_h^Y(p) = \frac{1}{2} \left(\cos\left(\frac{\pi p}{r}\right) + 1 \right), \quad (9)$$

where $d_h^X(x)$ is a X -axis horizontal deviation filter function, r represents the radius (kernel size), and $p = [-r, r]$ is the distance of a pixel to the kernel centre. The same notation is valid for Y .

4.3 Water to Object Interaction

We address four types of forces, all of which have similar implementation details: buoyancy, drag, lift and collision force.

Common feature of these forces is that they are computed for each face rendered from a top view orthographic camera into a texture. Blending must be turned on so that the information from some faces is not overridden. In order to efficiently sum the texture on the GPU and transfer only the result, we implement parallel reduction.

Buoyancy force In order to compute the buoyancy force we need to know the volume of the object's submerged part. The volume is obtained as the depth difference of the front and back faces of the rigid body. More specifically, it is obtained by an orthographic projection with blending and summed together.

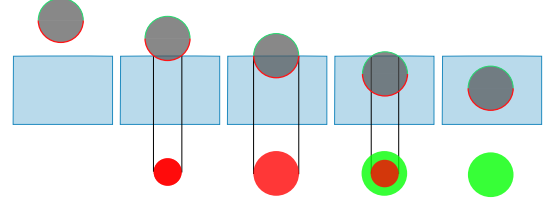


Figure 9: Upper image row represents a water tank seen from side view. We show different positions of a sphere relatively to the water level. Note that the sphere is green on the top (positive z -coordinate of the normal) and red on the bottom (negative normal). The bottom row shows the same object seen from a top view without the part which is above the water level.

Drag and lift force Drag and lift force [12] are computed for each face centroid and rendered into a texture similarly to the buoyancy force.

Water-object collision This force is calculated from the wave particle render texture. In addition of height value we also render wave particle propagation direction and speed in the remaining colour channels. Again we sum the texture in order to obtain the final force. While summing the forces we use buoyancy texture as an object silhouette stencil to precisely select which particles are affecting the object.

4.4 Object to Water Interaction

We have covered the process of the particle generated based on the wave particle distribution texture in section 4.1. This section denotes the distribution texture generation.

In order to create wave particle distribution texture we have to find out the silhouette of the submerged part of the floating object when looking from a top orthographic view and then distribute the wave effect (water volume) to the contour of that silhouette.

Figure 9 shows an example of such case. The silhouette is the union of the green and red part while the contour is the outer border of the red part.

The direction of the created wave is dependent on the direction of the object motion as shown in figure 10.

Important step of the object to water interaction stage is distributing the indirect wave effect to the object contour. This routine also smooths out the contour normals in order to uniformly cover the circular area around the object by the wave particle propagation angles.

Similarly to a parallel reduction approach we sum up neighbouring pixels into one. In each step we merge four adjacent pixels into one pixel. After few iterations when the texels are summed together the process is reversed and we reconstruct the original silhouette while using the textures from the intermediate steps. While descending to

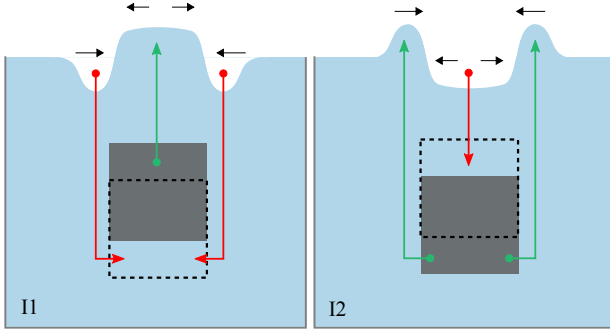


Figure 10: Different cases of wave propagation with respect to the position and the motion of the floating object. Striped line represents the object position in the previous time step. Cases I1 and I2 (inside) show the influence of both direct and indirect wave effect inside the volume.

higher texture resolutions we distribute the total indirect wave effect pixels recognized as contour pixels.

5 Results

To enhance the perception of the water surface, we render the surface with an approximative method [4] for light reflection and refraction. For reflection, the method uses a render of the flipped scene and modifies texture lookups according to the surface normal. We also use adaptive tessellation and tillable Perlin noise [6] as a height function to add high frequency details to water surface. Longitudinal waves are added in form of deformation of the x and z axis of the water plane similarly to Gerstner waves [11].

We have tested the performance of each simulation step in several testing cases.

Test case A In the first test case, particles are added to the buffer until it is full. This test case measures the performance of the wave particle propagation procedure. Note that in the wave particle method the number of particles is not directly proportional to the quality of visual result. Especially when most of the particles in the system have low amplitudes. We have compared the performance of the wave particle routine to the performance mentioned in the original article. They mention three test cases with different maximal number of wave particles. Since the wave particle generation routine is part of our wave particle propagation routine, we have measured both of those steps at once. Therefore, in our comparison we have also summed corresponding columns from the original article. Our test ran on the following configuration: Intel Core i5-4590 3.30GHz, GIGABYTE GTX970 4GB, 8.0 GB RAM, Visual C++ compiler 18.00. We also show the computing power of the processor used in the original article (3.19 Gflops [9]) compared to our processor (12.5 Gflops [9]) to demonstrate the hardware difference.⁴

⁴Both values are measured on the same benchmark test.

Implementation	10k	600k	8M
CPU approach[12] (2007)	1.430	3.87	200.04
our GPU approach	0.196	1.94	22.50

Table 1: Comparison of the wave particle method implementation with varying number of particles (ms).

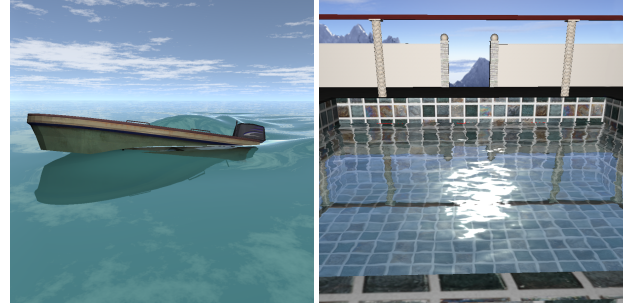


Figure 11: Test case A: ocean environment (left). Pool test case (right).

Test case B The second test case is a single boat floating in the ocean scene (figure 11). We created two modifications of this test case. In the first run we test the influence of the rigid body complexity on the performance of each phase. For example computation of drag and lift force is done per each face of the rigid body. Figure 12 shows the testing results. Simulations steps which are not dependent on the variable parameter are omitted.

Similarly we measured the influence of the wave particle render texture resolution (figure 12) on the performance. Note that this parameter affects not only the texture resolution, but also indirectly affects the number of fragment shader invocation etc.

Test case C The purpose of test case C is to show the usage of the wave particle method in a real simulation scenario. We have placed the boats in the scene in order to maximize the number of interaction between nearby floating object. Once the main boat moves, it creates wave front which pushes away the other floating objects. We measure the performance for different number of boats in the scene.

Test case D The fourth test case captures a scenario with high number of boats. Unlike the test case C, all the boats are moving.

Table 5 shows the average frame rates for test case C and D.

Ship count	1	2	4	8	16	32
Test case D	199	144	93	60	35	18
Test case C	145	136	97	60	33	24

Table 2: Average frame rate for the test case C and D.

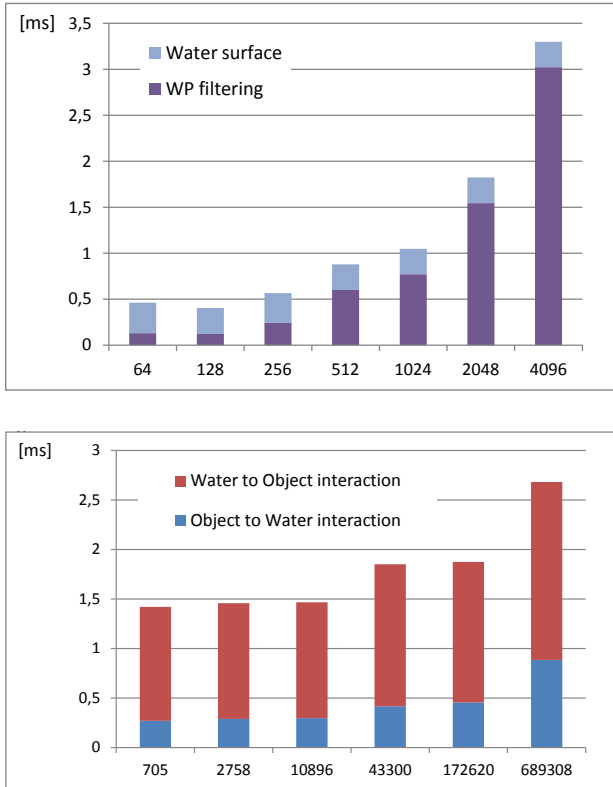


Figure 12: Mesh complexity (number of faces) influence on the simulation performance in ms (top). Particle render texture resolution complexity (bottom).

5.1 Limitations

We have developed such structure that each floating object contains its own render cameras, render textures, and wave particle buffers. This means that each boat in our simulation is independent and has every piece of localized information needed for the simulations. On the other hand, there is a structure using a single render texture with a single wave particle buffer shared for each floating object similarly to the implementation of Yuksel et al.

Our approach offers higher flexibility in terms of the floating object setup e.g. positions are not limited by the render texture resolution. Another advantage is that we



Figure 13: Test case C (left), and test case D (right).

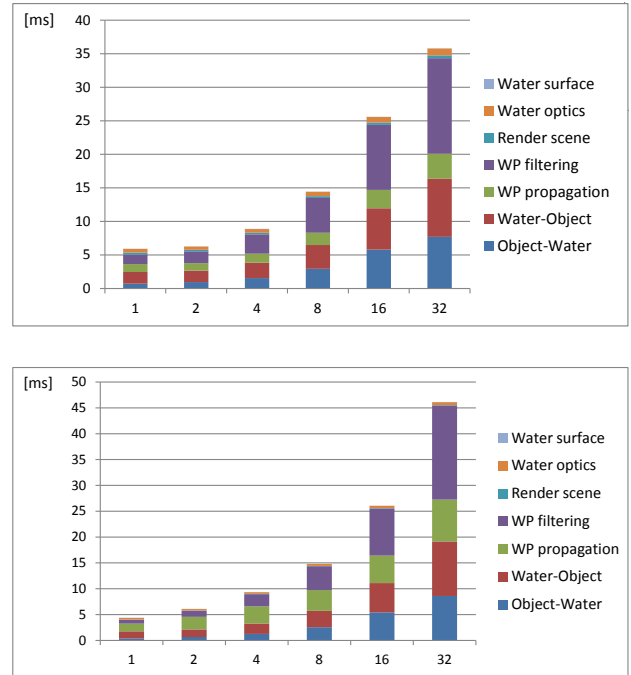


Figure 14: Result of the test case C (left), and case D (right).

can address only particles created by a concrete floating object, which is beneficial e.g. in LOD approach when we render only particles which come from a visible floating object.

On the other hand this approach has a limitation in the number of floating object in the simulation. Table 5 show significant performance drop for a large number of floating objects. Moreover, because we render the water surface in one step, all the vertical deviation functions are added at once to the global deviation. And since the number of GPU texture units is limited, we cannot apply all deviation function in one rendering step.

6 Conclusions

We have described theoretical background behind wave particle method for the purpose of real-time water simulation. We have showed how wave particles form continuous waves and that it can be used in an interactive environment. Consequently, we have implemented the wave particle method on the GPU. Part of our implementation is also the interaction between the water surface and a general shaped floating rigid body which can be controlled by user. We have measured and evaluated the performance of our GPU approach and showed that it offers plausible results at interactive frame rates.

References

- [1] specification of OpenGL version 4.40.

- [2] Haogang Chen, Qicheng Li, Guoping Wang, Feng Zhou, Xiaohui Tang, and Kun Yang. An efficient method for real-time ocean simulation. In *Technologies for E-Learning and Digital Entertainment*, volume 4469 of *Lecture Notes in Computer Science*, pages 3–11. Springer Berlin Heidelberg, 2007.
- [3] CT Chou and LC Fu. Ships on real-time rendering dynamic ocean applied in 6-DOF platform motion simulator. In *CACS International Conference*, volume 3, 2007.
- [4] Lund University Claes Johanson. Real-time water rendering : Introducing the projected grid concept, 2004.
- [5] E. Galin, J. Schneider (editors), H. Cords, and O. Staadt. Real-time open water environments with interacting objects, 2009.
- [6] John C. Hart. Perlin noise pixel shaders. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '01, pages 87–94, New York, NY, USA, 2001. ACM.
- [7] Geoffrey Irving, Eran Guendelman, Frank Losasso, and Ronald Fedkiw. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. *ACM Trans. Graph.*, 25(3):805–811, July 2006.
- [8] Juraj Onderik, Michal Chládek, and Roman Ďurikovič. SPH with small scale details and improved surface reconstruction. In *Proceedings of the 27th Spring Conference on Computer Graphics*, SCCG '11, pages 29–36, New York, NY, USA, 2013. ACM.
- [9] Primatelabs. Cpu benchmarks.
- [10] Karthik Raveendran, Chris Wojtan, and Greg Turk. Hybrid smoothed particle hydrodynamics. In *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '11, pages 33–42, New York, NY, USA, 2011. ACM.
- [11] Jerry Tessendorf. Simulating ocean water. in *simulating nature: Realistic and interactive techniques*. *ACM SIGGRAPH*, 2001.
- [12] Cem Yuksel, Donald H. House, and John Keyser. Wave particles. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)*, 26(3), 2007.

Arbitrary-Precision Arithmetics on the GPU

Bernhard Langer

Supervised by: Thomas Auzinger

Vienna University of Technology

Abstract

The majority of computer applications employ numerical data types with a fixed amount of precision for their computations. Their limited numerical range and precision are sufficient for most use cases. However, for some purposes, such as cryptography or geometrical computations, the required range and precision can become arbitrarily large. Numerical types that can handle such demands have higher memory requirements and are not natively supported by common hardware, which leads to increased computational complexity. In this paper, we examine how basic arithmetic operations on arbitrary-precision integers can be adapted to many-core architectures in the form of graphics processing units, which are widely available as commodity hardware. Apart from a detailed description of our method, we show superior performance characteristics of our implementation in comparison to state-of-the-art CPU libraries for high computational loads.

Keywords: gpu, cuda, integer arithmetics, parallel, arbitrary-precision

1 Introduction

Fixed-Precision Number Formats Arithmetic computations in most programs are performed using number formats with a fixed precision. These types allocate a constant amount of memory for each number to store its value and, therefore, only a limited amount of different values is available. A set of fixed precision formats is natively supported by common processing hardware, usually given by power-of-two binary lengths, e.g., 8-64 bits. Other arbitrary but fixed lengths have to be mapped to the hardware capabilities by software means.

Two main types of numbers can be differentiated: Fixed-precision *integers* are mostly used for counting or addressing purposes and are limited to a specific numerical range. Arithmetic operations on such numbers can result in an under- (resp. overflow), where the result of a computation is larger than the largest (resp. smaller than the smallest) possible value of the given data type. Fixed-precision *floating-point numbers* are used to represent approximations of real numbers and are limited both in precision and range. Consequently, rounding errors are a common downside, with implications depending on the application scenario.

Arbitrary-Precision Number Formats In some cases, fixed-length number types are not sufficient; for example, if the largest occurring value of an integer is not known prior to execution or if rounding errors of floating point arithmetics cannot be tolerated. In these cases, we can make use of arbitrary-precision number formats, for which the numeric range and precision are chosen dynamically. Arbitrary-precision arithmetics are essential to many applications, such as geometric algorithms or public-key cryptography [4].

Standard number formats are part of every major programming language, however only few of them provide arbitrary-precision number types (e.g., Lisp, Erlang, Java, Perl). For other languages, third party libraries have been developed to support such formats, such as the GNU Multi-Precision library (GMP) [8] or the Library for Efficient Data types and Algorithms (LEDA) [1]. Note that they explicitly target CPU hardware architectures.

Such computations are more complex when compared to regular hardware-supported 32/64-bit arithmetics. Basic addition/subtraction has a cost of $O(n)$ with n being the length in bits, while multiplication ranges between $O(n^2)$ and the conjectured optimum of $O(n \log(n))$ [5, 7, 10, 18].

To combat these performance issues, our overall goal in this paper is to leverage the capabilities of many-core hardware architectures to speed up arbitrary-precision computations. Specifically, we will make use of Graphics Processing Units (GPUs) by designing suitable data types and parallel algorithms. We present an implementation using a general and widely used GPU framework, the Compute Unified Architecture Framework (CUDA).

2 Related Work

Since General Purpose Computing on Graphics Processing Units (GPGPU) is a relatively new field, the majority of the work on arbitrary-precision arithmetics targets CPUs, which spawned several libraries. We already mentioned LEDA [14] and GMP [6] in the previous section and another established library is ARPREC [3] which itself is based on MPFUN [2], a multiple precision library for Fortran. Although many of them already provide a rich set of different data-types and operations, our goal is to accelerate the underlying computations for the use in time-critical applications. To our knowledge, there is no arbitrary-precision library for GPUs available and we

cover the existing works on fixed-precision arithmetics in the following.

GPU Multiple-Precision library (GPUMP) In 2010 Kaiyong Zhao and Xiaowen Chu created the GPUMP [23], a multiple-precision library for CUDA. GPUMP performs its operations on integer types with an arbitrary but fixed length. The functionality of GPUMP includes operations such as (modular) addition and subtraction, multiplication, division, Montgomery reduction/multiplication, exponentiation as well as comparators. GPUMP applies sequential arithmetic algorithms on pairs of numbers in parallel. It fails if the number grow beyond the predefined length limit and becomes inefficient for small numbers in terms of both computation time and memory usage. Since GPUMP is only applicable on integers with fixed length, its use in areas like geometry is very limited, whereas our work is based on arbitrary-precision integers.

Multi-Precision Floating-Point on GPUs A multiple-precision library for floating-point number types, the CUDA Multi-Precision library (CUMP), was presented by Takatoshi Nakayama and Daisuke Takahashi in 2011 [15]. Additional work has been done by Andrew Thall [22], Mian Lu et al. [13], as well as Mioara Joldes et al. [11].

2.1 Algorithms

In this section, we shortly review relevant parallel algorithms for our setting.

Parallel Algorithms in CUDA The use of parallel primitives on graphics hardware architectures was pioneered by Mark Harris and colleagues. We build on these concepts and refer to them [9, 19] for a detailed overview on the necessary considerations for algorithms to map well to GPUs and CUDA in particular, such as the usage of shared memory buffers, optimal memory access schemes and issues with code path divergence.

Integer Multiplication While integer addition is rather straightforward, their optimal multiplication is still an open problem. We use the standard school method with complexity $O(n^2)$. More advanced approaches, such as the divide-and-conquer approach by Anatolii Karatsuba [12] have lower complexity of $O(3n^{\log_2 3})$, the conjectured optimum of $O(n \log(n))$ is most closely reached by methods employing the Fast Fourier Transformation (FFT) [7, 18]. Such methods either show their advantage only for huge numbers ($>10^3$ decimal digits) or they are hard to efficiently map to graphics hardware. We show that our simple approach still runs significantly faster than current state-of-the-art CPU implementations.

3 Methodology

Arbitrary-precision arithmetics can be performed on various different number types such as integers, rationals or

algebraic. The fundamental number type is the unsigned integer type, additional signs can be handled separately. As arbitrary-precision rational numbers are usually composed of a sign and two integers, we target the unsigned integer type in this paper. Our arbitrary-length integer representation format is based on an array of unsigned sub-integers of fixed length, which we denote as *words*. The word length should be chosen to map well to the underlying arithmetic hardware and in the following, we assume that common arithmetic operations (e.g., $+$, $-$, \times) are natively supported on words. Furthermore, we expect such operations to ‘wrap around’ in case of an overflow, i.e., all operations are computed modulo the largest representable value of a word plus one. Note that this is the standard behavior for unsigned integers in all common languages. While theoretically unbounded, the amount of available memory will limit the number of words that can be stored and will act as a practical limit on the maximal size of our arbitrary-precision integers, which we will simply denote as *numbers*.

According to standard literature, the many-core processing hardware (i.e., the graphics card in our implementation) is referred to as *device*, while *host* refers to the CPU (plus the standard system memory). Furthermore, the part of the program executed on the device will be referred to as *kernel* [17].

3.1 Parallelization Strategy

While we target graphics hardware in particular, our work generalizes to most common many-core architectures (e.g. Intel Xeon Phi), which exhibit Single Instruction, Multiple Data (SIMD) computation units as their atomic elements. Each unit computes ℓ_{SIMD} -many data elements in parallel in each execution cycle. Our algorithmic design is strongly motivated by the observation that instruction divergences are costly if they happen inside a SIMD unit but incur no additional cost when different SIMD units follow diverging code paths.

A key assumption is that we expect the word count of the numbers, which we operate on, to be larger than the SIMD units’ length. For smaller numbers, the parallel extensions of CPUs (e.g., SSE or AVX) can be efficiently used. In our case, we employ a two-level parallelization strategy. First, we distribute each issued computation to one SIMD unit and compute them independently and in parallel. As different computations generally handle input numbers of different lengths and employ different operators ($+$, $-$, \times), significant instruction divergence is expected between them. All associated downsides are negated since SIMD units act independently from each other on our targeted architectures. Each SIMD unit itself operates on ℓ_{SIMD} -many words in parallel and we rely the provided intra-SIMD synchronization capabilities to handle sequential sections of the algorithms. After a computation is finished, the responsible SIMD unit fetches the next item from the computation pool until its depletion. Note that this approach is only efficient on large data set, where more computations than SIMD

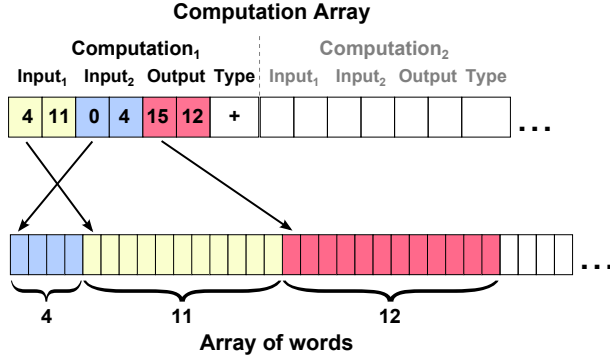


Figure 1: Memory layout of our numbers showing one example computation with two input and one output numbers as well as the operation type (+, −, ×) to be performed. The computation array holds the offset and size of the corresponding words in the global array of words.

units are issued.

3.2 Memory Management

A global array stores all words and the numbers can be identified by their offset into this array and their size (see Figure 1). A separate array holds all the computations that are issued. For each computation, the offset and length are stored for two input and one result number as well as the type of operation to be performed. As the offset and size of the results is generally not known in beforehand, we supply the functionality to reserve additional space in the global array of words. We keep a sophisticated memory allocator [20] as future work and just point to the first free memory location. Note that if host and device manage separate memory spaces (as is the case with graphic cards), data transfers have to be issued.

3.3 Addition

Sequential Addition Before moving to parallel algorithms, we first take a look on how two numbers X and Y are added by a sequential method. As already mentioned, the numbers are composed of several words, which we enumerate as x_0, \dots, x_m and y_0, \dots, y_n with x_0 and y_0 holding the Least-Significant-Bits (LSBs). We assume without loss of generality that $n \leq m$ holds for the word counts of the two numbers.

As addition is natively supported on words, we still have to account for potential carries. We will add each word-pair x_i, y_i sequentially and in case of an overflow due to the finite range of the numeric type of the words, we pass a carry c_i to the next addition. This can be achieved with a simple loop over all words. In each iteration, we compute the sum $s_i = x_i + y_i + c_{i-1}$. In case of an overflow, we rely on the wrapping behavior for s_i and issue a carry $c_i = 1$ (instead of $c_i = 0$) for the next addition. Note that for iterations $i > n$, we set $x_i = 0$ and terminate with the last iteration $i = m + 1$, where $x_{m+1} = y_{m+1} = 0$.

Parallel Addition with Word Counts $< \ell_{\text{SIMD}}$ While one can trivially add the corresponding words (i.e., each $x_i + y_i$) in a parallel manner, synchronization issues arise from the carry propagation due to its sequential nature. In this section and the next, we describe the addition of numbers whose word count is smaller than the width ℓ_{SIMD} of the SIMD units. We generalize for numbers of arbitrary length afterwards. All $m < \ell_{\text{SIMD}}$ additions of the terms are executed in parallel by a SIMD unit. Each addition potentially issues a carry that has to be propagated to the more significant words.

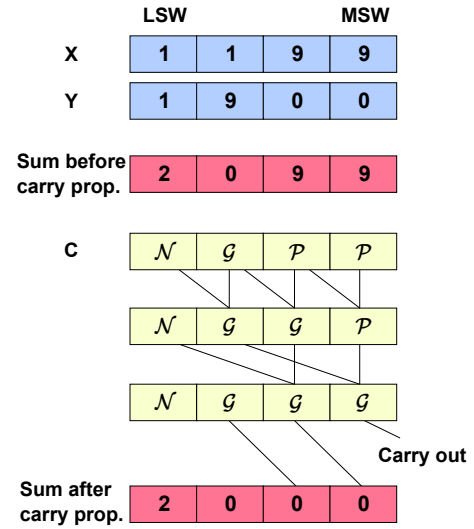


Figure 2: Illustration of the carry propagation with decimal number type as words: We compute the sum of numbers X and Y with lengths $n = 4$ for each word separately and perform the parallel carry propagation using the additional array C with values \mathcal{G} for generation, \mathcal{P} for propagation and \mathcal{N} for no carry. The numbers are ordered from Least-Significant-Word (LSW) on the left to Most-Significant-Word (MSW) on the right. The carries are propagated in $\log(n) = 2$ many steps and added to the sums to obtain the final result (bottom).

Parallel Carry Propagation To perform the carry propagation in parallel, we will use the prefix scan algorithm illustrated in Figure 2. Since we perform the addition of X and Y in parallel, we will store these carries in a temporary array C , where each addition $x_i + y_i$ can produce a carry that is stored in c_i . A carry c_{i-1} can only be propagated if $x_i + y_i$ is the maximum value v_{max} of a single word. A little convenient detail is that carry propagation and generation can not occur at the same time. Even if x_i and y_i are at the highest value, we have $v_{\text{max}} + v_{\text{max}} \bmod (v_{\text{max}} + 1) = v_{\text{max}} - 1$. Thus it is possible to store both cases (propagation and generation) in the same carry array C . We denote the three distinctive values in this array with \mathcal{G} for generation, \mathcal{P} for propagation and \mathcal{N} for no carry.

Now we have to find a generalized associative operation \otimes that can perform this propagation. Given a pair of

C_i	C_{i-1}	$C_i \otimes C_{i-1}$	C_i	C_{i-1}	$C_i \otimes C_{i-1}$
\mathcal{N}	\mathcal{N}	\mathcal{N}	\mathcal{G}	\mathcal{P}	\mathcal{G}
\mathcal{N}	\mathcal{G}	\mathcal{N}	\mathcal{P}	\mathcal{N}	\mathcal{N}
\mathcal{N}	\mathcal{P}	\mathcal{N}	\mathcal{P}	\mathcal{G}	\mathcal{G}
\mathcal{G}	\mathcal{N}	\mathcal{G}	\mathcal{P}	\mathcal{P}	\mathcal{P}
\mathcal{G}	\mathcal{G}	\mathcal{G}			

Table 1: Results for carry propagation function \otimes . \mathcal{G} for carry generation, \mathcal{N} for no generation and \mathcal{P} for possible carry propagation

carry values, it computes the resulting behavior and when iteratively applied to all pairs, it correctly adds carries to the relevant values. We list all possible combinations in Table 1. In the case that c_i is already set to \mathcal{N} or \mathcal{G} it does not matter which value c_{i-1} has as the initial value remains the same. In the last three cases, where c_i is set to \mathcal{P} , it will inherit the value from c_{i-1} , therefore \mathcal{P} is our identity element.

Parallel Addition with Word Counts $\geq \ell_{\text{SIMD}}$ For longer numbers, we can use a simple loop as described above for a sequential addition algorithm. The only difference is, that we do not iterate over every single word but instead over chunks of ℓ_{SIMD} -many words. Thus, each SIMD unit performs chunk-wise addition sequentially. For additional work parallel approaches are suggested.

3.4 Parallel Multiplication

We employ a parallel version of the school algorithm to multiply two numbers. Again, we assume two numbers X and Y , each composed of m and n words x_0, \dots, x_m and y_0, \dots, y_n . In contrast to addition, where the upper bound on the length of the result is $\max(m, n) + 1$, for multiplication it is $m + n$ and we thus store the result in an array P of words p_0, \dots, p_{m+n} .

Multiplication with Word Counts $\leq \ell_{\text{SIMD}}$ For simplicity, we will first take a look at multiplication of two numbers with a maximum length of ℓ_{SIMD} words each. Again, a single SIMD unit performs this computation, with all others running in parallel. The basic idea is to compute each line of the example on the right sequentially, while the workload of a single line is distributed across the processing elements of a SIMD unit. Note that in this example one word corresponds to one decimal place with 10 possible values. We start by multiplying x_0, \dots, x_m with y_0 and writing the result in p_0, \dots, p_{m+1} . Keep in mind that the first sub-product is of the length $\leq m + 1$. Then we compute the second sub-product of x_0, \dots, x_m with y_1 , which is added to the previous result but shifted by one word to the left, i.e., we add it to p_1, \dots, p_{m+2} using our

$$\begin{array}{r}
 5 \ 1 \ 2 \\
 \times 1 \ 2 \ 8 \\
 \hline
 4 \ 0 \ 9 \ 6 \\
 1 \ 0 \ 2 \ 4 \\
 5 \ 1 \ 2 \\
 \hline
 6 \ 5 \ 5 \ 3 \ 6
 \end{array}$$

addition algorithm from before. We will continue this until the last sub-product of x_0, \dots, x_m with y_n that will reside in the result array in p_{n-1}, \dots, p_{m+n} .

Sub-Products We now take a look at how to perform the j^{th} line of the example. Within the SIMD unit, each element i will perform the multiplication $x_i y_j$. Although we assume that the multiplication of two single words is supported, we cannot directly apply it, as the result will be two words long.

Alternatively, we will split the numbers x_i and y_j in two words of half size each, with x_{high} being the most significant bits and x_{low} being the least significant bits of x_i . Then we will perform four multiplications $x_{\text{high}} y_{\text{high}}$, $x_{\text{high}} y_{\text{low}}$, $x_{\text{low}} y_{\text{high}}$, $x_{\text{low}} y_{\text{low}}$ and store the (shifted) results in the two words p_{high} and p_{low} . Each processing element i adds its result p_{low} to the result array at p_{i+j} in parallel. After that, we perform a carry propagation. Finally, each element adds its result p_{high} to the result array at p_{i+j+1} where another carry propagation is performed.

Multiplication with Word Counts $> \ell_{\text{SIMD}}$ If we only increase the length of Y , the algorithm works just fine, since the limitation given by the SIMD length ℓ_{SIMD} only concerns the length of X . For longer X , we split it into chunks of ℓ_{SIMD} words each and process them sequentially. For the k -th chunk of X and the j -th word of Y , for example, the SIMD unit would compute the product $(x_{k\ell_{\text{SIMD}}}, \dots, x_{(k+1)\ell_{\text{SIMD}}-1}) y_j$.

4 Implementation in CUDA

In our implementation we mapped our parallel algorithms to CUDA [16]. The natively supported integer data type has 32 bits while the length of a SIMD unit – called *warp* – is also 32. Thus, both our word length (in bits) and SIMD length ℓ_{SIMD} are set to 32, making a SIMD-sized chunk 1024 bits long. Carry propagation was performed with intra-warp prefix scans using shared memory, while result space reservation employed global and shared memory atomics. At kernel start, we spawned as many warps as the device supported in blocks of integer size and terminated them only after the computation pool was depleted. No intra-block synchronization primitives were used as we rely on the implicit intra-warp synchronization.

5 Results

For different test cases, we compare the timings of code execution on the CPU with the timings on the GPU on a test system with an Intel Core i7 4700MQ CPU and a nVidia K2100M GPU. We organize the test cases according to computation type (+, −, ×) and the amount of computations that are issued. The length of our numbers are randomly sampled from a normal distribution with mean μ

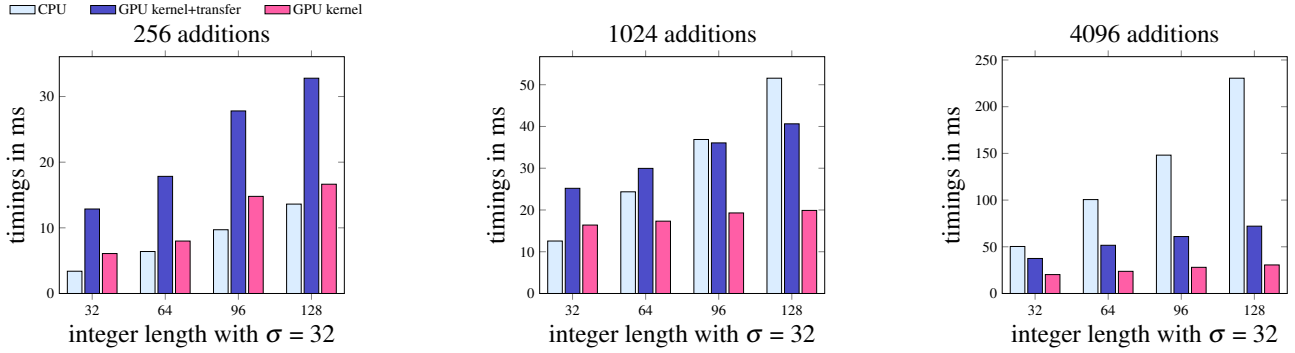


Figure 3: Addition benchmarks. Integer lengths in multiples of 1024 bits with deviation σ are shown on the x-axis. The timing of our GPU implementation is shown with and without data transfer to and from the device on the y-axis. For small computational loads, the GPU is not sufficiently occupied, while for a sufficiently large amount of computations, superior performance is obtained.

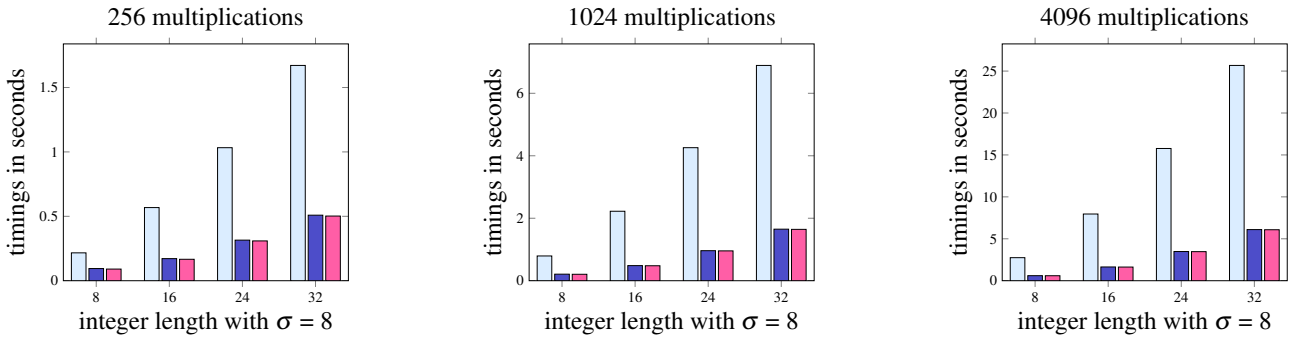


Figure 4: Multiplication benchmarks. Integer lengths in multiples of 1024 bits with deviation σ are shown on the x-axis. Due to the higher complexity of the multiplication, the GPU outperforms the CPU already at 256 multiplications and the advantage grows slightly as we raise the computation count.

and standard deviation σ to reveal possible code divergence issues.

Benchmarks We compute our benchmarks for the operations $+$, $-$ and \times . For each of these operations we instruct the GPU and CPU to perform a predefined number of operations. We create a pool of 1024 randomly generated large integers sampled from the standard distribution with μ and σ and each operation is performed on two randomly chosen numbers from this pool. The benchmark results shown in the plots are the averaged timings of 32 executions. The GPU benchmarks are computed with our own framework, while we generate the CPU benchmarks with the state-of-the-art LEDA library [14] as an objective reference and leave optimizations on the CPU as future work.

Addition and Subtraction Comparison The operation types addition and subtraction perform almost identically, since they use the same algorithms and the conclusions in this section hold for both operations. In the first test case with only 256 operations (see Figure 3), the CPU computations are still performed faster compared to our own framework or almost equal if we do not take the data transfers into account, since the computational load is too

small and the GPU not fully occupied. As we increase the amount of operations to be performed, we can see an advantage of the GPU – data transfer taken into account – at around 1024 operations. The advantage of the GPU and the performance gap between CPU and GPU increases with every raise of the operation count as we saturate the full compute capabilities of the graphics hardware.

Multiplication Comparison Due to the higher computational complexity of multiplications, we already see the performance advantages of the GPU compared to the CPU at the first test case with 256 operations, although the lengths of the integers are only a fourth of the lengths in the additions and subtractions benchmark. Due to the shorter numbers used for the multiplication benchmarks, the data transfer happens relatively fast, and the two cases with and without transfer behave the same. Already at 256 operations, the GPU performs around three times as fast as the CPU and this performance gap almost remains throughout our test cases, as shown in Figure 4. At 4096 operations the GPU performs about four times as fast as the CPU.

6 Limitations and Future Work

As our work is a pioneering effort into arbitrary-precision integer arithmetics on graphics hardware, there are multiple venues for future work:

Faster Multiplication One could replace our school-method multiplication with the Karatsuba Algorithm [12] for a better computational complexity. Even better performance on longer numbers can be achieved with the use of FFT based methods [21].

Parallelization Strategies For small numbers, a per-thread parallelization can yield better device occupancy for a smaller amount of computations. For huge numbers, a per-block or per-device parallelization can lead to better occupancy as well.

Additional Formats A support for rational numbers as quotients of two integers would add implicit division capabilities. However, an efficient method to compute the greatest common divisor would be needed to reduce the memory requirements. Furthermore, the framework can be extended to support algebraic number formats, which is highly non-trivial due to the conceptual and algorithmic complexities involved.

Additional Operations Although the framework is a proof of concept, it can be extended to make it practically usable. For that it needs to support more mathematical operations than simple arithmetics, such as exponential functions, least common multiple, greatest common divisor, min/max functions and comparators.

7 Conclusion

We presented a method to perform arbitrary-precision integer arithmetics on massively parallel hardware in the form of graphic cards. By employing a two-level parallelization scheme, we ensure minimal code divergence within the SIMD units while still providing effective load balancing across all units. By employing parallel prefix sum computations we allow for an efficient carry propagation and dynamic computation of memory offsets to both read and write integers of arbitrary length. Our CUDA implementation was compared to a state-of-the-art CPU-based libraries and with several benchmarks we showed that method is several times faster for large computation loads.

References

- [1] Algorithmic Solutions. *The LEDA User Manual*, version 6.4 edition, July 2012.
- [2] David H. Bailey. MPFUN: A portable high performance multiprecision package. Technical report, NASA Ames Research Center, December 1990.
- [3] David H. Bailey, Hida Yozo, Xiaoye S. Li, and Brandon Thompson. ARPREC: An arbitrary precision computation package. *Lawrence Berkeley National Laboratory*, 2002.
- [4] Joshua Davies. *Implementing SSL/TLS using cryptography and PKI*. John Wiley and Sons, 2011.
- [5] Anindya De, Piyush P. Kurur, Chandan Saha, and Ramprasad Saptharishi. Fast integer multiplication using modular arithmetic. *SIAM Journal on Computing*, 42(2):685–699, 2013.
- [6] Laurent Fousse, Guillaume Hanrot, Vincent Lefevre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13, 2007.
- [7] Martin Fürer. Faster integer multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009.
- [8] Torbjörn Granlund. *The GNU Multiple Precision Arithmetic Library*. GMP development team, 6.0.0 edition, March 2014.
- [9] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.
- [10] David Harvey, Joris Van Der Hoeven, and Grégoire Lecerf. Even faster integer multiplication. *arXiv preprint arXiv:1407.3360*, 2014.
- [11] Mioara Joldes, Valentina Popescu, and Warwick Tucker. Searching for sinks of Henon map using a multiple-precision GPU arithmetic library. *HAL archives*, November 2013. 6p. <hal-00957438>.
- [12] Anatolii Alexeevich Karatsuba. The complexity of computations. In *Proceedings of the Steklov Institute of Mathematics – Interperiodica Translation*, volume 211, pages 169–183. Providence, RI: American Mathematical Society, 1995.
- [13] Mian Lu, Bingsheng He, and Qiong Luo. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 19–26, New York, NY, USA, 2010. ACM.
- [14] Kurt Mehlhorn. *LEDA: a platform for combinatorial and geometric computing*. Cambridge University Press, 1999.
- [15] Takatoshi Nakayama and Daisuke Takahashi. Implementation of multipleprecision floating-point arithmetic library for GPU computing. In *Proceedings of the 23rd Int. Conf. on Parallel and Distributed Computing and Systems*, IASTED '11, pages 343–349, 2011.
- [16] NVIDIA. CUDA, 2014. <http://www.nvidia.com/cuda/>.

- [17] nVidia. CUDA toolkit documentation v6.5. <http://docs.nvidia.com/cuda/index.html>, 2014.
- [18] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3-4):281–292, 1971.
- [19] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.
- [20] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *Innovative Parallel Computing (InPar)*, 2012, pages 1–10, May 2012.
- [21] Daisuke Takahashi. Parallel implementation of multiple-precision arithmetic and 2,576,980,370,000 decimal digits of π calculation. *Parallel Comput.*, 36(8):439–448, August 2010.
- [22] Andrew Thall. Extended-precision floating-point numbers for GPU computation. In *ACM SIGGRAPH 2006 Research posters*, page 52. ACM, 2006.
- [23] K. Zhao and X. Chu. GPUMP: A multiple-precision integer library for GPUs. In *10th International Conference on Computer and Information Technology*, CIT '10, pages 1164–1168. IEEE, 2010.

Visualization & Applications

OpenGL View Library

Adam Riečický*

Supervised by: Martin Madaras†

Faculty of Mathematics, Physics and Informatics
Comenius University in Bratislava
Bratislava/Slovakia

Abstract

In the paper, we propose a library for the viewing of OpenGL textures, models and other resources. The included library is adding a possibility for users to open an additional window beside their program, which can be used for displaying models, variables and textures in multiple ways. The user can configure window layout and customize what will be displayed in the window. The library also supports creating more layouts and switching between them during runtime. The user is free to apply his own shaders and vertex attributes for individual objects to customize the rendering. The library can be used as a supportive viewer or tool for debugging OpenGL applications.

Keywords: OpenGL, mesh viewer, texture viewer, C++ library, buffer visualization

1 Introduction

Nowadays, many programmers need to use graphical output in their programs. It may become useful also for the programmers for whom the graphical output is not the main intention. For this purpose, various graphical libraries such as OpenGL [1] or DirectX are chosen to get maximal rendering efficiency.

Since these libraries are working directly with graphic accelerators, the performance is robust, but it has its cost especially for the developers. Data stored in video card memory are hard to review by the program debugging and errors on this level are unpleasant and disturbing. Moreover, programmers often have these data stored on graphic card related with data in their application, which makes debugging even more frustrating.

Our goal was to create a visualization tool for these programmers, where they would be able to inspect models and textures stored in the video memory, and connect it with the data from their program. This should be done by providing a library with a simple interface which can create another window beside the user's application and offer previewing possibilities.

The paper is organised as follows. The first section is devoted to similar solutions. It provides a closer look on currently existing tools, and summarizes what is missing in those solutions compared to our solution. The second part specifies how the rendering of the structures is done. The next section is devoted to a rough description of the implementation. The last section summarizes the testing and results that we have achieved. A concrete outcome of this work is a library that can monitor and display different types of data for various applications.

2 Related Work

The most widely used debuggers which we discuss and compare in this section are gDEBugger [2], nSight [3], Vogl [4] and few others. Each of them offer slightly different features, but the main purpose is the same, similar with ours.

The gDEBugger is being promoted as an advanced OpenGL debugger, profiler and memory analyser. It is a stand-alone application, which allows the user to select an executable he wants to debug. It runs the executable under its environment and allows the user to display textures, shaders, OpenGL state and other resources created and used by the application. After pausing the application, the user can list down all textures, buffers, and objects located on the graphic card in a moment. It also informs the user about the performance and function calls. Since gDEBugger is running over an executable, the user cannot debug the variables from his application. Our library can display both memory and video-memory data, also beside the custom debugging environment.

In contrast, the nSight debugger from Nvidia extends traditional code debugging environments. It can be built in the Visual studio or Eclipse environment and offers the user an ability to see almost anything what is related to the video card memory, while he is debugging his own code. The nSight can be used for debugging on CPU, GPU and shaders simultaneously. However, an obvious main disadvantage of this product is that it supports the latest versions of the nVidia graphic cards only. Our solution is designed to support also older versions of OpenGL.

The Vogl is being promoted as an OpenGL capture / playback debugger. It is a new tool currently in alpha re-

*a.rieicky@gmail.com

†martin.madaras@gmail.com

lease, which support both Linux and Windows platforms. It handles logging all OpenGL state into the file, with reviewing possibilities. Compared to our solution, at the current state it does not offer any graphical output for debugging, despite the large amount of information logged.

There are also many other tools which can be used for debugging OpenGL contexts. The BuGLE [5] - similar to nSight but running on UNIX-like systems. It can be used for debugging and profiling OpenGL applications including shader code, buffers and a visual feedback of the textures, the color and the depth buffers. Since November 23 2014, BuGLE is no longer being developed. Another tool, the GLIntercept [6] can log all OpenGL calls but it was mainly designed for OpenGL to version 2.1. Can be declared that it is similar, however older solution then the Vogl.

Despite all the pros of the mentioned tools, there are many limitations. None of the tools provides a visualization of meshes, and that was the main reason and motivation to create our own. We wanted to let the user to see not only the array of values in the buffer, but also a 3D visualization of them. Debugging the meshes in a visual form is much more intuitive then listing a buffer values.

Most of the tools, like gDEDebugger and nSight, need to pause the application before they can be used. It can be restrictive in some cases, for example when debugging an animation, and it may lead to complications such as need of frame-by-frame data debugging. Our library works real-time, allowing the users to view mesh or texture animations instantly.

In mesh processing algorithms, an output is often represented as an array of values, corresponding to individual vertices of the mesh. These values can be for example mesh diameter in vertex, skinning weights for single bone, curvature or other vertex properties. Rendering of these values in the user's application would need creation of specific shaders and buffers applied on a meshes. We wanted to offer the user a possibility for creating a link between data in the memory and data on the video card, with a visual output. Our solution transforms an array of values into a color data and then assign them to a vertices of a mesh, which is the feature that is not present in any of known debuggers.

In order to allow the user other possibilities, we added a visualization of texture data and displaying variable values. To sum up, our tool may not be as complex as the existing solutions mentioned before, but it provides visualization possibilities that are beyond the limits of the other tools.

Compared to other debugging tools, our framework is displaying data defined by the user only, instead of all OpenGL context. This may result in the better clarity of displayed data, and filtering all not required buffers.

Similarly to the gDEDebugger, our framework works in an additional window running beside the users application. This window displays all resources monitored by the tool and can be customized, depending on the needs and pref-

erences of the user. This feature also makes it suitable for supplementary visual output applications, not only for debugging purposes.

3 Visualization Methods

Our solution is displaying user defined structures only. We needed a mean to uniquely identify them, which would also help the user to distinguish between them. All the data monitored by the library have therefore its unique caption, defined by the user.

3.1 Mesh Visualization

Mesh rendering is the feature of our framework on which we were focused the most. In its simplest form, our framework can render a vertex buffer displaying a mesh as a point cloud. This form of visualization does not require additional information about the mesh, beside a buffer and number of vertices. This allows the user to get a visual feedback in a single library function call. In the next step, the mesh data can be adjusted by other callings to specify vertex connectivity, texture coordinates or vertex attributes.

The vertex connectivity can be added by sending an index buffer. Specifying element type, viewer treats a vertex array as a sequence of elements. From that moment the mesh is not rendered just as a point cloud, but individual connected elements can be seen. Similar procedure can be used for the objects which should be textured, but there is a need to have the vertex connectivity specified in the moment. Object is then displayed as a full solid textured mesh, which can be viewed in the window.

After this, a rendering of the model is done by the library's internal shader program which can be replaced by the user specific shader program. It can modify the way how the model in a library window is rendered. It is possible for the user to use vertex attribute buffers for a shader input as well.

To each model, an array of values can be assigned. It manipulates the vertex colors depending on the value. For each individual vertex one value is taken from a field and transformed to a color using a selected color scale. There are several possibilities how the value can be changed into the color. For our solution we selected a linear color mapping function, which is trivial to implement, with a low computing cost, and result which is adequate and fully sufficient for our needs. The vertex value is mapped on a predefined linear color scale (e.g. blue for the minimal and red for maximal value), and then applied as a vertex color.

3.2 Other Data

Beside the one main purpose, which is a vertex-buffer and mesh visualization, we want to offer the user ability

to display other structures, to enlarge usage possibilities. Specifically there are two other options - inspection of the textures and tracking application variables.

Textures stored in the video memory can be monitored via their individual buffer ID generated by OpenGL. Sending this ID to the library allows the user to display specific textures in the Viewer window. All the textures that were sent to the library, can be displayed and viewed. This feature is not as complex as the mesh viewing, but we decided to implement it, because it often may come in handy to have it available. It can be used for example to review depth or color frame buffer textures, normal, diffuse and other texture properties of a model.

The last function allows the user to set a variable pointers to the library and then display actual values. Output of the each pointer can be formatted and inserted into a defined string line. These strings can then be selected in a library window and are displayed as standard text output. This function may be useful when there is a need to see variables in a real-time without the restriction of application pausing.

4 Implementation

The library offers a set of tools for previewing data structures. User can specify which OpenGL context he wants to share with the library, and which structures he wants to display. Viewing framework then makes a list of these structures and the user can select and see the actual look of the data at runtime.

Primarily, the library is designed to work with OpenGL version 4+, however it can be used with projects that are written in lower OpenGL version standards. The project is implemented as a static library that can be linked to any C++ project. The library header file, contains all function callings that the library provides.

4.1 Architecture

The library interface function callings can be used to specify which resources should be displayed in the library window. By using them, the user passes to the library all information needed. The framework stores all the data which has been sent and offers them to select at runtime (User-Library interaction described in Figure 1). Each structure has its own caption - a user defined text description of the resource. The user can identify and select the resource he wants to display by the caption.

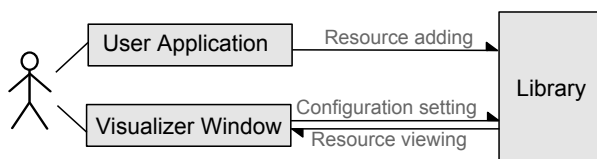


Figure 1: User - Library interaction

The Viewer Library runtime is separated into two modes to make it easier to adjust for different purposes. First, there is the configuration mode. It allows the user to create a layout of the window and specify parts of it, where the rendering of the individual structures will be done. This layout serves as a starting point for the second, and more important part - the viewing mode.

The viewing mode is the main feature of the program, which allows the user to list all stored data, such as meshes and textures stored in video memory or to display variables. It is possible to select in real-time what to display at the current moment. The number of structures and data that can be reviewed in this mode are dependent on the code interface calls. That means, that if there were no interface calls, nothing can be viewed in the viewing mode.

4.2 Context Sharing

Each application that is running OpenGL has its context. It stores all the state associated with the instance of OpenGL. Each resource generated and stored on the graphic card is specific for the instance of the context, which means that two applications running OpenGL cannot see nor access the context of the other.

Our Viewing Library uses context sharing. This means that on initialization, the context created by the user program needs to be shared with the library. It allows the library to operate on the same context as the user program. The difference between separated and shared context is shown in Figure 2.

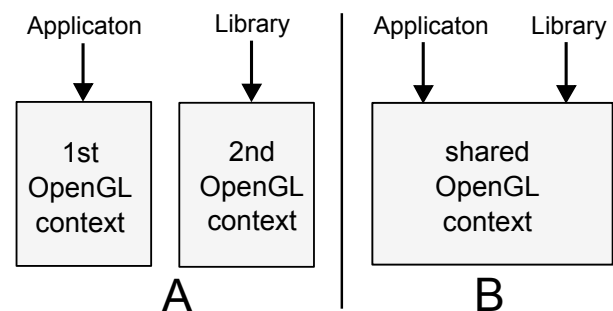


Figure 2: A - two separate OpenGL contexts, B - single shared OpenGL context

4.3 Viewing Mode

The viewing mode is the main part of the application, and it is the actual visual feedback for the user. In this mode the user can display all the data which were sent to the Library through the code. In Figure 3, an example of viewing layout is displayed.

Viewing mode distinguishes three types of the field, which can be rearranged in the configuration mode. Mesh view is used for displaying vertices from the vertex buffers and seeing the correlation of them with the value arrays

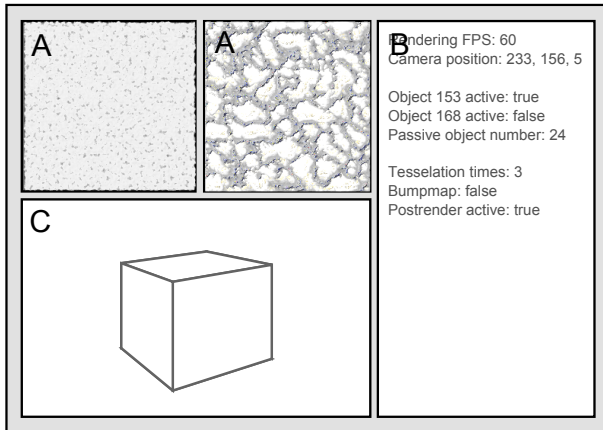


Figure 3: Example of the viewing mode layout.
A - Texture views, B - Variable display, C - Mesh view

from the computer memory. It is possible to rotate and zoom the individual objects in the view. Texture view determines an area where the textures can be displayed. Variable view is the area which displays text output and the actual value of the defined variables. Variables are added by the user program, and lines to display can be selected by the user for each individual variable display.

4.4 Configuration Mode

This mode is focused on creating a custom layout for the window, as the one which can be seen in Figure 4. The layout is represented as a set of rectangular fields. The type of each can be changed by the user. There are three main types of fields which can be selected: mesh, texture and variable field type.

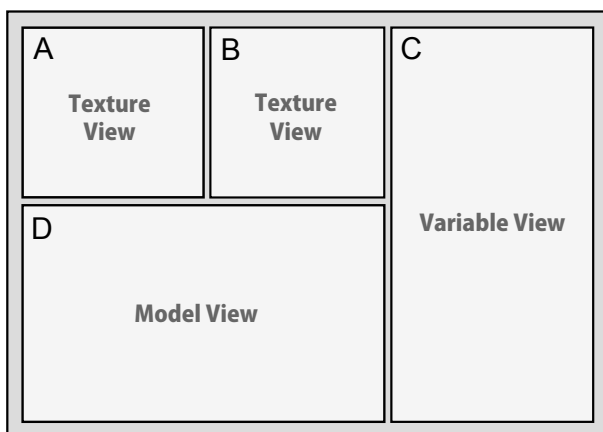


Figure 4: Example of the configuration.

Our framework supports profile creation. Each profile can hold different configurations of a window. The user can switch between profiles and modify them at runtime. Each session comes with one default profile which cannot

be deleted and which can be selected any time. All other profiles can be freely renamed or deleted.

All changes made in the configuration mode are applied to the current profile. These changes are automatically saved. During the next creation of the Viewer window the previously created layout is loaded, therefore there is no need to configure the window layout every time the program is started.

4.5 Usage Example

To demonstrate the usage of our framework, let's assume an example. The user wants to preview two loaded models as a point cloud, a texture and a variable which holds number of renders of his application.

First step is including the library into the project and calling the initialization function at the start of the program. Then for each resource he needs to send a buffer and his own description of it. It is one function calling for each of the models, and one for the texture. Finally, he uses another function to specify output string and set a pointer to the value which represents the number of renders. When he now runs an application, an additional window pops up beside his application window (if there is any). This window is currently empty and there is no possibility to see anything yet. Currently there is *configuration mode* running, which means that the user can set up a layout. He creates a layout which consists of each mesh, texture and variable field, and then switches to *viewing mode*. Once the mode is switched, resources can be displayed and previewed in the fields.

5 Results

To test all the features of the library we run it on an application used for model manipulation and computation. This application provides mesh processing algorithms and calculations on meshes. One specific feature of the program is the calculation of a Shape Diameter Function [7] for graphs, which can nicely demonstrate the use of the mesh and data linking in the library. The application is also working with other data, such as generated textures and variables.

In Figure 5 the tested application and the window of our Viewer running on top of it can be seen. Textures which were generated by the client application are instantly sent to the library. These textures can be displayed in the Viewer. Figure 6 shows a similar test on a different model. Both Figures show that the vertices of the model are colored in the Viewer window. These colors are a visual representation of the Shape Diameter Function values, which is defining the diameter of the mesh in an individual vertex. The values were calculated by the application, sent to the Viewer as a pointer to an array and assigned to the model by its caption. Our conclusion to this test was that

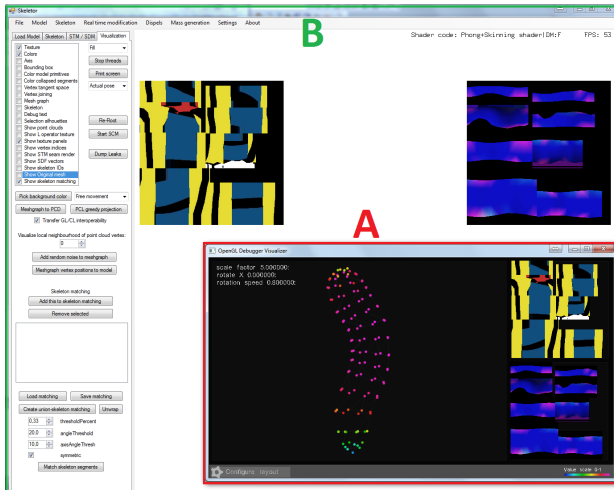


Figure 5: Viewer window (A) used beside an application (B).

the Viewer is correctly and tabularly displaying all data we sent to it.

Images indicate that the Viewer provides sufficient visual feedback for the user who is developing such an application. Without a need of the users own rendering environment, the visualization possibilities of the tool can be easily used to display several kinds of resources.

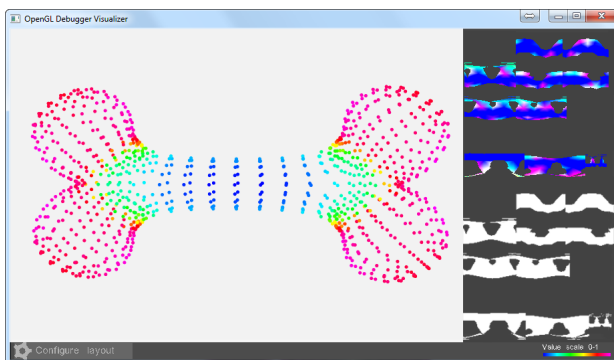


Figure 6: Model with applied values of Shape Diameter Function and its displacement and height map.

Since the tool provides multiple visual output options, there are many possible applications for it. For example it may become useful when the programmer needs to determine if some mesh/texture was loaded correctly. Using just a few commands it allows the user to have a visual feedback of a mesh or a texture, where he can inspect it and determine its correctness.

Another application can be a visual test of a shader program. For one user specified model there is a possibility to apply a shader program. The model can be rendered in the Viewer window with the shader. The user can see whether the render behaves as it should, or there are some undesirable artefacts. The user is also free to add another shader and the same model, and able to see both renders

and compare them.

6 Conclusions

Our goal was to create a library which can display meshes with their supplementary data, textures and variable values on a separate window which can run beside the user's application in real-time. We reviewed similar existing tools, but since we were not able to find any solutions for exactly this type of problem, we have been inspired by existing debuggers for OpenGL, which were closest to the subject of our work.

Our library can be further expanded by adding more customization possibilities for the fields, user interface upgrades like texture zooming or functional expansions such as printing a matrix values. It can be used for viewing OpenGL and the user-program variable arrays. The library can be useful in several applications, mainly as an alternative tool for displaying OpenGL resources, debugging a program or just an auxiliary viewer running in a detached window.

References

- [1] The Khronos Group. Opengl 4 reference pages. <http://www.opengl.org/sdk/docs/man/>, 1997-2015. [Online; accessed 19-February-2015].
- [2] Graphic Remedy. gdebugger. <http://www.gremedy.com/>, 2004-2011. [Online; accessed 02-January-2015].
- [3] NVIDIA Corporation. Nvidia nsight. <http://www.nvidia.com/object/nsight.html/>, 2015. [Online; accessed 15-March-2015].
- [4] RAD Game Tools Valve Software. Vogl. <https://github.com/ValveSoftware/vogl>, 2015. [Online; accessed 15-March-2015].
- [5] Bruce Merry. Bugle. <https://www.opengl.org/sdk/tools/BuGLE/>, 2007-2014. [Online; accessed 10-March-2015].
- [6] Damian Trebilco. Glintercept. <https://code.google.com/p/glintercept/>, 2003-2012. [Online; accessed 10-March-2015].
- [7] Lior Shapira, Ariel Shamir, and Daniel Cohen-Or. Consistent mesh partitioning and skeletonisation using the shape diameter function. *Vis. Comput.*, 24(4):249–259, March 2008.

Visualizing Archaeological Excavations based on Unity3D

Thomas Trautner*

Supervised by: Gerd Hesina†

VRVis Research Center, Vienna / Austria

Institute of Computer Graphics - Vienna University of Technology, Vienna / Austria

Abstract

As part of an archaeological excavation, huge amounts of different types of data, for example laser scan point-clouds, triangulated surface meshes, pictures or drawings of finds, find attributes like location, age, condition and description or layers of excavated earth are collected. This detailed documentation is important to give archaeologists the possibility to analyze the collected data at a later date, since the find spot might not be accessible anymore. Unfortunately, all the accumulated data is separately saved and consequently complex to explore.

Therefore, we present a novel solution that allows the user to digitally explore a virtual archaeological excavation in real-time. With our approach, we can not only visualize different types of textured meshes and finds, but also allow the user to draw on surfaces to mark areas of certain interest that need further exploration, enable explosion views to investigate composition of different layers of earth and arbitrary slicing of the three-dimensional mesh structure to better visualize cross-sections, and an easier tracing of accumulation points of finds. The result of this work is a new powerful tool that will support the analysis of future excavations. All results and the implementation itself will be presented as part of this work.

Keywords: Unity3D, Archaeology, Visualization

1 Introduction

This work complements the already implemented *Harris Matrix Composer* [a], abbreviated below as *HMC*. The HMC is an important tool archaeologists use for the documentation of archaeological sites. Our 3D viewer is an additional component of the new *Harris Matrix Composer - Plus* system. The viewer is a real-time renderer with a GUI shown as a toolbox which allows the user to further explore and verify a given three-dimensional data set. This allows the user to visualize selections and/or combinations of all types of stratigraphic units within the HMC.

The HMC is based on the *Harris Matrix* which was invented in 1975 by Edward Harris [8]. During an excavation, this matrix is used to document the stratigraphic

relations of dug out sediments. Every unit of stratification – for example remains of wood or brick walls, a basement, or inclusions – is displayed as a single node in the hierarchical graph. Figure 1 shows a typical Harris Matrix.



Figure 1: Example of a Harris Matrix within the Harris Matrix Composer. It is a two-dimensional and graph-like representation of dug out stratigraphical layers. Additionally we highlight a subdivision which represents room 1 of the excavated ruin in Falkenstein.

A typical HMC matrix starts with a circular green unit at the top. It represents the top surface of the archaeological excavation, for example measured by a laser-scan. The lowest unit is as well a circular green unit, which stands for the lowest excavated stratigraphic layer. Every other unit in-between consists of at least a top- and a bottom-surface, or a hull of these two surfaces. Additionally, a HMC matrix enables an accurate assignment of the find locations. Every path visualizes a different location, and therefore a subdivision of the excavated area can be easily identified.

Unfortunately, beside all the already mentioned advantages of a two-dimensional graph-like representation, domain experts from the field of archeology (like the *Ludwig Boltzmann Institut für Archäologische Prospektion und Virtuelle Archäologie* [c]) came to the conclusion that it is necessary to explore not only a graph-like representation, but also the real excavated three-dimensional data set. This is essential to better understand an excavation and easily track possible mistakes made during the docu-

*thomas.trautner@tuwien.ac.at

†hesina@vrvis.at

mentation process – like, for example, wrong classification or age determination of layers.

Therefore we extended the HMC and implemented this 3D viewer which allows archaeologists to (1) compare a *Harris Matrix* with measured three-dimensional data (2) perform manipulations on the data in real time and further (3) verify assumptions and findings. Doing this without our 3D viewer is extremely difficult and time-consuming, because all layers of earth are irretrievably removed during an excavation.

2 Related Work

A very similar approach of visualizing archaeological excavations is presented by J. Cosmas et al. [4]. The authors propose a tool to visualize building parts, finds, stratigraphical layers and textures in situ, and furthermore provide advanced 3D reconstruction techniques. This is done by storing all different types of data in a multimedia database. The result is used for presentation or publication purposes.

Another approach presents a virtual reality tool [2] which allows the user to measure stratigraphical units like curvature, length, thickness, height or volume to enhance archaeological fieldwork. Furthermore it allows the user to calculate volumes and simulate archaeological formations and deformation processes which is essential for understanding an excavation.

The work of Vote [13] focuses on visualizing the Great Temple of Petra based on post-excavation archaeological analysis. As part of this work they analyzed four different prototypes. First of all a conceptual model that automatically assigns chronology by using finds and other datable objects, secondly a three-dimensional database and a *Geographical Information System* (GIS) visualization software, thirdly visualizations using a virtual-reality interface and finally they focused on improving visual perception using lighting and coloration.

Dellepiane et al. [5] present a technique of interactive slicing, where a slicing plane is moved further away toward the viewing direction to visualize disparity between two different time steps – a technique that is also very useful for archaeological visualizations.

Benko et al. [3] focus on a mixed-reality approach implemented for multiple users. It supports the tracking of see-through head mounted displays and multi-touch table surfaces. The final visualization is similar to our approach based on the *Harris Matrix*.

Additionally, we considered a summary [9] of five papers that propose different approaches to visualize seismic measurements. This is done to allow the user to further explore and analyze possible oil or gas reserves. For optimal use the author recommends the following expressive visualization and rapid interaction techniques: the time of creation of such an illustration must be reduced, the program must independently recognize important stratigraph-

ical layers, and the exploration should be as simple as possible for the user.

Remondino and Campana [10] present image-based approaches to capture detailed 3D information of archaeological excavations. This is done with cost-efficient approaches like taking pictures of finds from different angles, and finally computing the three-dimensional structure. Therefore we implemented a run-time object loader for our HMC-Plus 3D viewer which is able to load and visualize such models as well.

A similar low-cost approach is presented by Doneus et al. [6]. Instead of expensive techniques like measurements with laser scanners, they use a technology called *Structure from Motion* [12] and create a three-dimensional point cloud. This approach requires only minimal technical knowledge and user interaction, and is therefore straightforward to use.

Allen et al. [1] introduce a 3D Modeling Pipeline for archaeological excavations and finds. First, the excavation site is either scanned by a laser scanner or pictured with cameras. In the next step, the site is represented as 3D model, and additional context like background images, videos or GIS data is added.

Santos et al. [11] focus on realistic rendering of archaeological excavations. They concentrate on illumination methods like global illumination. The paper presents different approaches that guarantee a frame-rate of at least 10 images per second and the possibility to change the view point dynamically. Since lighting is such an important factor for a realistic perception of a scene, we allow the user to dynamically place light sources in the scene with our HMC-Plus 3D viewer.

To provide stereoscopic rendering we further implemented a technique called off-axis rendering, which was described by Grasberger [7]. His work presents different approaches of stereo rendering and explains in detail their use, the advantages and disadvantages.

3 Implementation

Our main goal was building a tool that allows the user to scientifically visualize archaeological excavations. The resulting 3D viewer should be dynamic, customizable, easily extendable and has to support the import of high-resolution geometry and very large geometric models. Therefore, we chose the Unity3D Game Engine [b] as an optimal framework for our implementation. Furthermore, Unity 3D offers the possibility to build a 3D viewer for different platforms.

3.1 Object Import and Data Preparation

We wanted our rendering framework to be as independent and compact as possible. Therefore, we implemented a 3D viewer and partially extended it with already existing libraries that were well-suited for our approach – like for

example an improved file browser dialog [d] that allows us to display different folder structures including all their files during runtime. By additionally applying a filter function that allows the user to select only files with an (*.obj) extension, we managed to optimize the complex handling of the numerous 3D files. We furthermore expect the following naming restrictions to guarantee a successful mesh import:

[0-9] - [000-350] tbs_ [0-99] - [A-Za-z0-9]? .obj
[Sub-Mesh] - [Name] - [Room] - [Information].obj

Example:

0_000tbs_0_falkenstein_smart.obj
0_000tbs_1_falkenstein_smart.obj

Sub-Mesh: We use this index to detect if a mesh consists of smaller sub-meshes. In the given example the two objects are named 000tbs and belong to the same sub-mesh with index 0. If one of them is selected by the user during a load object operation, the other one will automatically be loaded as well.

Name: This name is displayed if the mouse cursor is moved over the mesh in our rendering window. The name consists of three digits that represent the layer and the acronym *tbs* which stands for *top-bottom surface*. The lowest layer of excavation should always be called 000tbs. All other layers can be called according to their HMC-label, although the highest possible number is currently 350tbs. This limitation results from the current maximum number of tags that are assigned dynamically during runtime. Unfortunately, tags must be hard-coded and cannot be created depending on variables which are passed during run-time.

Room: The room number is important to enable future versions of the HMC-Plus 3D viewer to perform manipulations only on individual rooms of the currently selected mesh (for example only explode layers of a single room). The current version of the 3D viewer explodes all layers equally independent from the selected mesh and room it belongs to.

Information: This part is not required by the 3D viewer. It can be used to store additional information assigned by the user. If there is no need for further information, it can be left out.

Unfortunately, our external triangulation library [e] and Unity3D itself have a maximum vertex count limit of 65.534 vertices per mesh. If such a big mesh is imported into the editor, Unity3D will automatically divide it into default sub-meshes. After building the project, or during runtime, this additional processing step is not possible anymore. Therefore, we expect the user to either load smaller meshes or divide meshes which have more vertices into sub-meshes manually in a preprocessing step.

3.2 Mesh Selection

If a layer is selected by clicking the left mouse button, the outline of the selected object is changed to red, which can be seen in Figure 2.

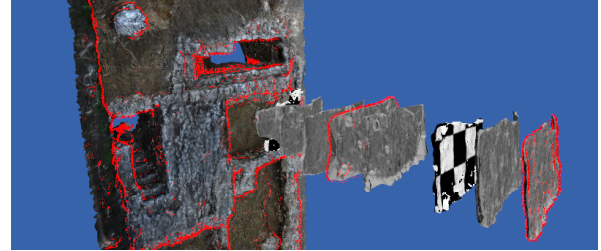


Figure 2: An example scene where three different meshes were selected. Their shader is changed from a diffuse shader to a red outline-shader. This feature allows the user to easily see which layers are currently selected.

As we can see the mesh is still textured, but the shader is changed from a simple diffuse shader to an outline-shader with red as its main color. If the "Ctrl" button is pressed and another object is selected, this object is selected as well, and their shader is changed to an outline shader. In Figure 2, three different layers were selected: First of all the 000tbs ground layer, then layer 074tbs in the middle, and the highest and therefore latest layer 006tbs on the rightmost side of the screen. If the "Ctrl" button is not clicked, the first selected object will be deselected after another element is selected. By clicking on the background all currently selected meshes will be deselected.

Especially this feature will be essential for future versions of the HMC-Plus 3D viewer. It will allow the user to select stratigraphical layers within the 3D viewer and directly get their position in the two-dimensional Harris Matrix.

3.2.1 Functioning of the Outline Shader

The result of the dot product is a scalar which represents the cosine of the angle between two vectors. If the dot product of the current viewing vector and the surface normal is zero, we simply change the pixel color to the outline color. If the dot product is not zero, we do a texture look up for the current pixel and calculate the dot product of the surface normal and the light direction to illuminate the pixel with a diffuse illumination model. An example of this algorithm can be seen in Figure 3.

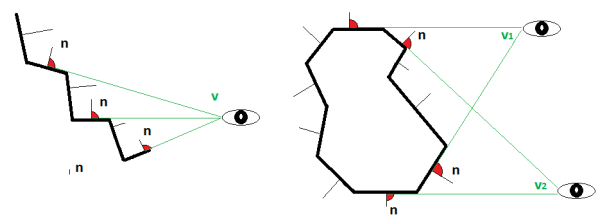


Figure 3: Low resolution meshes with an outline-shader

In the case demonstrated in Figure 3, the black polygon line represents an example layer which has an outline shader attached to it. Every triangle of this mesh has a surface normal which is represented by the vectors called n . The current camera position is represented by the eye with the corresponding viewing vectors called v . If the dot product is zero, the surface color will be changed to red, which is represented by the red angle in the middle. In this case vector n is orthogonal to vector v .

In Figure 3, we visualize two special cases: the cross-section of a triangulated surface without a correct red outline and the cross-section of a concave polygon with two viewpoints from which a correct outline is drawn.

3.3 Show Finds

During the excavation process, usually different items or materials like coins, glass, wood, porcelain, iron or others are found. To provide interactive exploration of find locations, our 3D viewer is capable of visualizing them as well. An example of this can be seen in Figure 4.

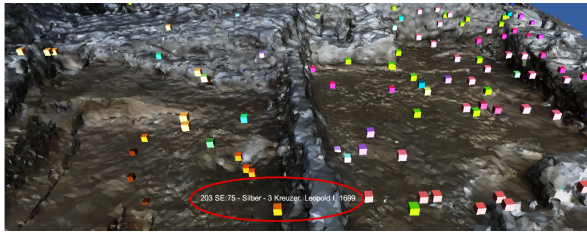


Figure 4: Example of finds of an excavation. They are represented as colored boxes whereas the color depends on the material properties of the find. Furthermore an additional description is shown if the mouse is moved over a box (for example year dates, the material, or layer number in which it was found).

Finds are usually represented by a point-cloud file which stores all X-, Y- and Z-coordinates (*.obj) and an additional description file (*.txt), which includes the layer number where it was found, the material and an additional description for every find. As soon as the 3D viewer has imported all finds, colored boxes will be displayed. Every box represents a single find.

Currently the color is chosen randomly, but finds with similar materials are always colored the same: For example, every silver coin is colored red, and every copper coin blue. To guarantee that the colors are significantly different and the same color is not selected twice, we used a hash-function that uses all the available find-information as seed to calculate the color. To improve the comparison of finds across different excavations, the next version of our 3D viewer will allow the user to select certain color maps to be able to use the same colors for materials which were already used in previous models of archaeological sites and only define new colors for new materials.

Furthermore, the user can change the color distribution, so that all finds of a certain layer will get the same color.

This feature allows the user to easily see which finds were found in which layer and where accumulations may be.

3.4 Slice Surface-Layers

Stratigraphy is a meaningful subsection of archaeology. Therefore it is important for archaeologists to understand how layers have moved, changed and sedimented over the years. Knowing this allows for example the age determination of finds. Unfortunately layers are often not well separated and therefore the Harris Matrix is not expressive enough. Therefore we implemented another important feature which allows the user to explore the structure of all stratigraphical layers of an excavation in explosion view. After the user has selected a layer, the 3D viewer will automatically calculate the bounding box of the selected mesh, and a slicing plane is shown. This is depicted in Figure 5.

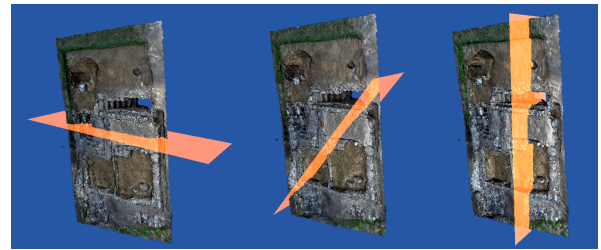


Figure 5: Different positions of the slicing plane

The dimension of the slicing plane depends on the dimensions of the bounding box. To guarantee that the slicing plane is always large enough to cut through the whole currently selected layer, the greatest dimension of the bounding box determines the width of the slicing plane. The user can then change the position and orientation of the slicing plane. In some special cases, like for example a diagonal position of the slicing plane, the greatest dimension of the bounding box is not enough. Therefore, we will increase the width and use the greatest distance between two points of the bounding box in the next version of the 3D viewer.

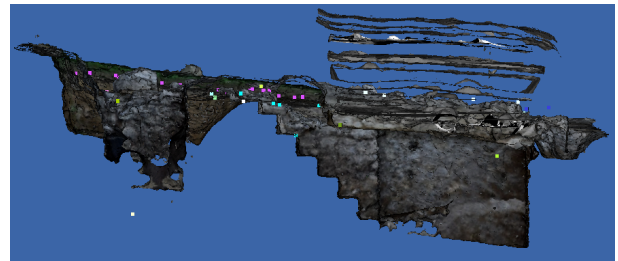


Figure 6: In this Figure we present a possible slicing view of an excavation. To visualize cross-sections of surfaces our 3D viewer uses orthographic cameras. Furthermore it is still possible to visualize finds and enable the explosion mode.

Additionally, we want the user to be able to place the plane exactly at the desired location. To simplify this task,

the slicing plane is semitransparent, and its color differs significantly from typical soil colors to further strengthen the contrast. The slicing view uses two orthographic cameras to visualize cross-sections of surfaces. An example of the sliced model can be seen in Figure 6. All the finds are visible and room 1 is currently in explosion mode.

3.5 Explosion View

Another important feature we implemented allows the user to explore the structure of all stratigraphical layers of an excavation in explosion view. Using this type of visualization, the hierarchical order and composition of all layers are better visible and therefore easier to analyze. The resulting animation steps can be seen in Figure 7.

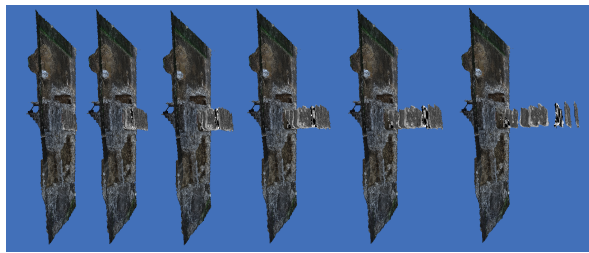


Figure 7: *Explosion view animation sequence*

Figure 8 shows a graphical explanation of our algorithm. The *expander* variable is a float value which controls the factor of explosion. Depending on the total number of layers called *k*, it is multiplied with *k* minus the current layer index.

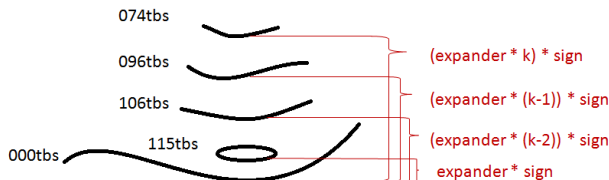


Figure 8: *Simplified explosion view explanation*

Therefore the collection of expandable layers is always sorted in descending order. The sign factor depends on the direction in which the mouse wheel was turned. With this adaption the same algorithm can either perform an explosion to enlarge the offset or an inverse explosion and move all layers back to their original position.

3.6 Brushing

The brushing tool can be used to mark certain regions of interest. This is important to identify future excavation sites, potential errors, overlooked finds and further to be able to easily export and exchange this information globally. To enable this, we implemented multilayer-texturing using an additional alpha texture. If our 3D viewer imports a mesh it will have two texture layers. The first layer

is the texture that the material file was pointing at. The second layer will be added by the viewer. It is a simple alpha-texture that is used for all the brushing operations. After selecting a color and enabling the brushing tool, the user can use the left mouse button to draw on a surface and the right button to erase already made drawings. To remove drawings, they are changed back to the original alpha value. The brush size is calculated dynamically, but currently only quadratic kernels are supported. Another characteristic can be observed in Figure 9: If the brushing is performed repeatedly on the same spot, the opacity of the color increases.

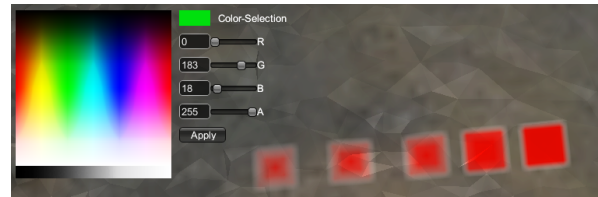


Figure 9: *Color selection and kernel with different opacities*

In the top left corner of Figure 9 the color selection [f] is visible. It can be used to select RGBA colors either by simply clicking on a color, or by inserting values from 0 to 255 for every red, green, blue and alpha channel.

After the brushing operation the alpha texture can be exported and for example imported the next time the users opens his model. It will be saved as (*.png) in the folder for textures and screenshots of the opened scene. In addition, the date and time is recorded in the name of the file. For example:

6-18-2014_10-42-50_AM_Texture_44

[month]-[day]-[year]-[hour]-[min]-[sec]-[AM/PM]-
Texture-[sequence number].png

An example of how marked regions could look like can be seen in Figure 10.



Figure 10: *Example of marked regions*

3.7 Adding Light Sources

To enhance the perception of the structure of the reconstructed geometry, the scene can be illuminated manually during runtime. The current version of the 3D viewer supports point lights; however, future versions will include

different types of light sources like for example spot-lights or directional-lights. Once a light source is added to the scene, its intensity can be changed. An example of how different intensities could look like is shown in Figure 11.

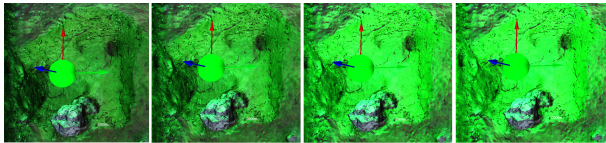


Figure 11: Controlling the intensity of a point-light source using the mouse-wheel

To manipulate the light source position three arrows representing the X-, Y- and Z-axis are displayed. By clicking on one of them with either the left or right mouse button, the position can be moved up or down along this axis respectively.

3.8 Name Highlighting

If the mouse cursor moves over a mesh, we automatically display the name of the mesh. We do this by continuously emitting rays from the mouse position into the viewing-direction. If such a ray hits a mesh-collider, we print a label with the corresponding mesh name next to the cursor. Thereby we allow the user to interactively explore the given data set without having to look up every layer within the Harris Matrix.

4 Additional GUI-Settings

Apart from the already mentioned exploration features, we allow the user to change some basic settings of the rendering system as well. These adjustments include the actual type of camera and the used render mode.

4.1 Camera modes

Currently we support two different camera modes: The first mode, which can be seen in Figure 12, is called *Fly-Through-Mode*. The second mode, which is visible in Figure 13, is called *Origin-Rotation-Mode*.

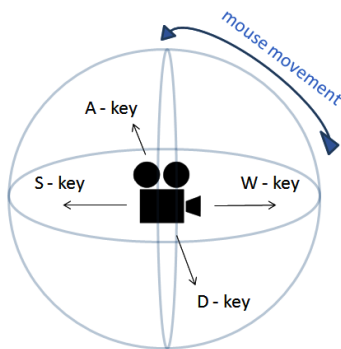


Figure 12: Fly-Through-Mode explanation

4.1.1 Fly-Through-Mode:

The viewing direction can be changed by moving the mouse, and the camera position can be changed by using the W, A, S and D keys.

4.1.2 Origin-Rotation-Mode:

Using this mode the camera moves circularly around its center-point. The distance to the center-point can be changed by using the mouse-wheel.

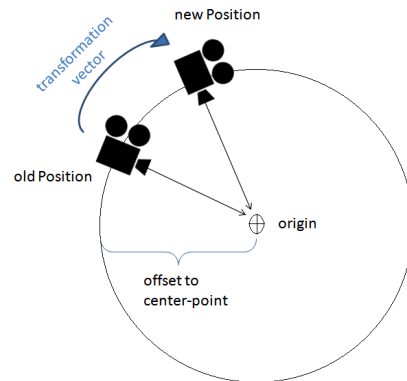


Figure 13: Origin-Rotation-Mode explanation

Additionally, this mode consists of two cameras to provide stereoscopic rendering. To render these images, we implemented off-axis-projection by using different projection matrices.

5 Results

The most important performance criteria for us was the applicability in real-time to allow the user to interactively explore our visualization. Therefore we focused mainly on GPU and CPU consumption and the average number of frames per second in this section. The following hardware settings were used on the testing computer:

Operating system: 64-bit Windows 7 Home

RAM: 16 GB

CPU: Intel(R) Core(TM) i7 -3930K CPU @ 3.20Ghz

Graphic card: NVIDIA GeForce GTX 690 with 4096 MB memory

The scene we used for testing consists of twelve different stratigraphic layers. Every layer is a triangulated and textured high-resolution point-cloud (*.obj). The biggest layer we used is the 000tbs ground layer which consists of 314.347 vertices and has an overall size of 20,17 MB. Furthermore, we used a 1024x1024 picture (*.png) taken during the excavation to texture the ground layer and a 128x128 checkerboard pattern (*.png) to texture all other sub-meshes. An overview of all object and texture sizes can be seen in Table 1.

Number of layers	1	2	3	4	5	6	7	8	9	10	11	12
Added layer	000tbs	006tbs	020tbs	032tbs	035tbs	054tbs	074tbs	096tbs	100tbs	106tbs	109tbs	115tbs
Vertices in layer	314347	6070	10931	10543	7734	10589	10393	8793	3067	10403	5731	3465
Vertices in scene	314347	320417	331348	341891	349625	360214	370607	379400	382467	392870	398601	402066
Size of *.obj file(s) (kB)	20.170	329	618	800	706	616	612	519	162	593	395	311
Texturesize of layer (kB)	2.370	4,74	4,74	4,74	4,74	4,74	4,74	4,74	4,74	4,74	4,74	4,74

Table 1: Overview of the example scene *Falkenstein*

Number of layers	0	1	2	3	4	5	6	7	8	9	10	11	12
Added layer	-	000tbs	006tbs	020tbs	032tbs	035tbs	054tbs	074tbs	096tbs	100tbs	106tbs	109tbs	115tbs
lowest FPS	140	133	120	103	88	80	73	69	43	35	31	22	19
highest FPS	173	168	165	152	149	148	132	113	109	91	80	75	60
average FPS	156	148	142	135	134	125	115	95	73	48	45	41	40

Table 2: Overview of the lowest, highest and average frame rate

Usually, an average frame rate of at least 35 frames per second is needed to ensure interactivity in real-time. Table 2 shows a minimum average frame rate of 40 FPS. The lowest frame rate occurs usually during and immediately after the importing of a new layer. Thereby we can guarantee that our viewer is able to deal with large data sets with a high vertex count.

6 Conclusions

The HMC-Plus 3D viewer was already presented to the potential end-user group of archaeologists of the *Ludwig Boltzmann Institut für Archäologische Prospektion und Virtuelle Archäologie* [c]. They found the 3D viewer to be especially helpful and easy to use – on one hand to watch cross-sections and explosion views to analyze structures and their relations, and on the other hand the stereo rendering for public presentations such as press conferences.

Although we built a stand-alone viewer and could therefore not totally benefit of a fully developed game engine like in our case Unity3D, we were able to achieve excellent results. We presented the development of a powerful tool, which is designed to simplify the future work of archaeologists. With this solution, it is possible to not only analyze a graph-representation of an archaeological excavation, but visually represent all measured data at run time. This provides a virtual exploration for archaeologists who were not able to physically visit the excavation site. In addition, it allows archaeologists to research different stratigraphic layers at the same time. This is usually difficult, as younger layers, which are mostly located above older layers, must be irretrievably removed to reach lower surfaces. With our approach no information is lost, and even younger surface layers can be studied easily.

7 Future Work

The goal of our future work will be to develop an interface between this viewer and other parts of the Harris Matrix Composer - Plus system. It will then be possible to not only investigate the three-dimensional data set, but also instantly determine its position in the Harris Matrix and vice versa. As result, we want to create a single, even more powerful tool out of already developed components, which can be seen in Figure 14.

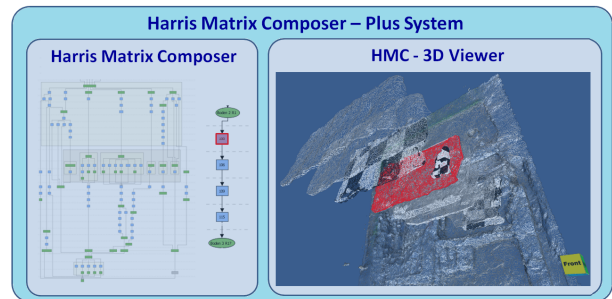


Figure 14: The final Harris Matrix Composer - Plus System will consist out of already implemented components for example the HMC and the 3D viewer. If a unit in the HMC is selected it will be automatically highlighted in the 3D viewer.

We further plan to replace the box-placeholder of the finds with their exact 3D representations. In addition we want to apply the Explosion mode not only on the stratigraphical layers but also to the finds, develop other intuitive camera modes, and implement ambient occlusion to improve the spatial perception for the users. Furthermore we plan to implement selection based manipulations like exploded views which only affect certain areas or rooms of an excavation.

Finally we want to expand our user study to ensure that our HMC-Plus system supports most of the archaeological work and techniques.

8 Acknowledgement

At this point I want to thank Michael Schwärzler from the *VRVis Research Center* for reviewing and always suggesting helpful improvement.

Furthermore we want to thank *Ludwig Boltzmann Institut für Archäologische Prospektion und Virtuelle Archäologie* [c] which generously provided us with actual data from their excavation in *Falkenstein*.

This work has been supported by a grant from the *Austrian Science Fund (FWF): P24597-N23 (VISAR)*.

References

- [1] Peter K. Allen, Steven K. Feiner, Alejandro Troccoli, Hrvoje Benko, Edward Ishak, and Benjamin Smith. Seeing into the past: Creating a 3d modeling pipeline for archaeological visualization. *Proceedings of the 2nd International Symposium on 3D Data Processing*, 2004.
- [2] Juan A. Barcelo, Oscar de Castro, David Travet, and Oriol Vicente. A 3d model of an archaeological excavation. *The Digital Heritage of Archaeology*, Computer Applications and Quantitative methods in Archaeology, 2003.
- [3] Hrvoje Benko, Edward W. Ishak, and Steven Feiner. Collaborative mixed reality visualization of an archaeological excavation. *Workshop on Collaborative Virtual Reality and Visualization (CVRV)*, 2003.
- [4] John Cosmas, Take Itegaki, Damain Green, Edward Grabczewski, Fred Weimer, Luc Van Gool, Alexy Zalesny, Desi Vanrintel, Franz Leberl, Markus Grabner, Konrad Schindler, Konrad Karner, Michael Gervautz, Stefan Hynst, Marc Waelkens, Marc Pollefeys, Roland DeGeest, Robert Sablatnig, and Martin Kampel. 3d murale: A multimedia system for archaeology. *Proceedings of the 2001 conference on Virtual Reality, archaeology, and cultural heritage*, pages 297–306, 2001.
- [5] Matteo Dellepiane, Nicolo DellUnto, Marco Callieri, Stefan Lindgren, and Roberto Scopigno. Archeological excavation monitoring using dense stereo matching techniques. *Journal of Cultural Heritage*, 2012.
- [6] Michael Doneus, Geert Verhoeven, Martin Fera, Christian Briese, Matthias Kucera, and Wolfgang Neubauer. From deposit to point cloud - a study of low-cost computer vision approaches for the straightforward documentation of archaeological excavation. *Geoinformatics CTU FCE 2011*, 2011.
- [7] Herbert Grasberger. Introduction to stereo rendering. *Student Project, Institute of Computer Graphics and Algorithms - Vienna University of Technology*, 2008.
- [8] Edward C. Harris. The stratigraphic sequence: A question of time. *World Archaeology*, 7, 1975.
- [9] Daniel Patel. Expressive visualization and rapid interpretation of seismic volumes. *Thesis for the degree of Philosophiae Doctor (PhD) at the University of Bergen - Norway*, 2009.
- [10] Fabio Remondino and Stefano Campana. Fast and detailed digital documentation of archaeological excavations and heritage artifacts. *Proceedings of the 35th International Conference on Computer Applications and Quantitative Methods in Archaeology (CAA)*, 2008.
- [11] Luis Paulo Santos, Vitor Coelho, Paulo Bernardes, and Alberto Proenca. High fidelity walkthroughs in archaeology sites. *6th International Symposium on Virtual Reality, Archaeology and Cultural Heritage VAST*, 2005.
- [12] Shimon Ullman. The interpretation of structure from motion. *Proceedings of the Royal Society of London*, 1979.
- [13] Eileen Louise Vote. A new methodology for archaeological analysis - using visualization and interaction to explore spatial links in excavation data. *Thesis for the degree of Philosophiae Doctor (PhD) at Brown Computer Science - Rhode Island*, 2001.

Links

- [a] **Harris Matrix Composer**,
June 25th, 2014 - 3:40pm
<http://www.harrismatrixcomposer.com/>
- [b] **Unity3D Game Engine**,
July 31st, 2014 - 10:30am
<http://unity3d.com/>
- [c] **Ludwig Boltzmann Institut für Archäologische Prospektion und Virtuelle Archäologie**
June 25th, 2014 - 3:15pm
<http://archpro.lbg.ac.at/>
- [d] **GUI - File Browser by Daniel Brauer**
May 26th, 2014 - 2:20pm
<http://wiki.unity3d.com/index.php?title=ImprovedFileBrowser>
- [e] **Object Loader by Jon Martin**
May 30th, 2014 - 3:45pm
<http://www.jon-martin.com>
- [f] **GUI - Color Picker by Sergey Taraban**
May 23rd, 2014 - 10:35am
<http://staraban.com/en/simple-color-picker-control-for-unity/>

Unity Hyperlinked Interactive Digital Storytelling

Irfan Prazina*

Supervised by: dr Selma Rizvić[†]

Faculty of Electrical Engineering Sarajevo
Bosnia and Herzegovina

Abstract

Virtual presentation of cultural heritage is significantly enhanced through the interactive digital storytelling. The common approach is to access the digital stories by clicking the interactive nodes within the virtual environment. We introduce a different method which enables the users to interact with the story and branch to hyperlinked digital content, which can contain virtual environments. The hyperlinked structure implementation is done in Unity. Evaluation of the method is performed through the user feedback analysis. **Keywords:** Interactive digital storytelling, hyperlinked stories, virtual cultural heritage.

Keywords: Digital Storytelling, Hyperlinked Video, Unity

1 Introduction

Storytelling is one of the oldest concepts in history of mankind. Stories are hidden in every aspect of human life. The methods of storytelling are changing and adjusting to the available media and tools. However, the aim remains the same: engage the consumer with the story and make him/her immersed in different space and time. With the development of digital technologies, the presentation of cultural heritage is significantly improved. Nowadays we can virtually travel through time and visit the 3D models of monuments in their original shape. Many tools and techniques are involved in such presentations, one of them digital storytelling. The museums are improving their physical exhibitions adjoining the artifacts with digital stories about their purpose, history and related events and characters. Today the people are used to all kinds of interactions. The time becomes a precious commodity. Very few people read books, most just skim through web sites and can afford to watch only short videos. The hypertext principle has conquered our everyday lives. The aim of this research is to explore how the perception of stories can be enhanced dividing them in short sub-stories hyperlinked in a hierarchical structure. On the example of the Tašlihan application we will discuss the user perception of this concept and its advantages and drawbacks in comparison with the lin-

ear storytelling. The paper is organized in the following way: Section 2 presents the existing concepts of hyperlinked storytelling structures such as hyper-video, in Section 3 we present the overview of our research in this field and introduce the new method of Unity hyperlinked video, Section 4 describes the case study we used as a proof of concept, in Section 5 we evaluate our concept through a user survey and Section 6 presents our conclusions.

2 Related work

One of the most common concepts of hyperlinked story structures is the hyper-video. It was first demonstrated by the Interactive Cinema Group at the MIT Media Lab. Elastic Charles [1] was a hypermedia journal developed between 1988 and 1989, in which "micons" (video footnotes) were placed inside a video, indicating links to other content. Following the Storyspace project, a hypertext writing environment, the HyperCafe, an award-winning interactive film, places the viewer inside a virtual cafe. It is a video environment where stories unfold around the viewer [2]. After these first works, and a rather long period of stagnation, many different methods of hyper-video implementations started to appear with development of Internet, starting in 2010s, most of them for use in advertising and marketing. Nowadays there are several popular tools using hypervideo. In the RaptMedia [3] cloud based editor, the user can create interactive videos and controls are implemented in form of links on the web. The Madvideo tool [4] is used to add tags to video files. Interactivity is implemented via manually inserted interactive tags. The tags can be links to websites, images, or other video clips. In the Open Hypervideo project [5] the contents are linked using annotation-types, such as: Wikipedia Articles, locations, videos, web pages, etc. Video sequences are made out of multiple (cut) video files. In E-Learning-How-Tos [6] the learning process via videos is enhanced using elective contextual data inside the videos. Cacophony, the interactive player for HTML 5 and JavaScript [7] allows creating interactive elements inside videos like story adapting in response to the user input. ClickVID video players [8] allow creating 'hotspots', clickable regions with specific content at designated time. WebM is a video file format made for HTML5 video tagging. Apart from the mentioned fields of application, the hyperlinked storytelling is

*iprazina1@etf.unsa.ba

[†]srizvic@etf.unsa.ba

used in virtual cultural heritage applications. A Human Sanctuary is a project implemented by the Cyprus Institute, telling the story about the famous Dead Sea Scrolls, with text annotations which offer more details about certain notions mentioned in the video [9]. In the Keys to Rome exhibition [10] the interactive digital storytelling was used to present the reconstructed Roman remains from Rome, Amsterdam, Alexandria and Sarajevo in combination with physical museum exhibits. Most of the mentioned projects use HTML5 and JavaScript as tools to make interactivity in the linear video sequence. Our idea is to try to use the Unity 3D game engine to connect the hyperlinked stories and combine them with interactive virtual models.

3 Interactive digital storytelling methodology

"Digital storytelling is narrative entertainment that reaches the audience via digital technology and media." In [11] Miller states that digital storytelling techniques can make a dry or difficult subject more alive and engaging to the viewers. In order to improve the classical storytelling concept, Glassner defined interactive storytelling as a two-way experience [12], where "the audience member actually affects the story itself". Manovich introduces the possibility for audience to change the story and offers the concept of an interactive narrative as "a sum of multiple trajectories through a database" [13]. We started our research of interactive video with the interactive video virtual tours [14], where the user is watching video walks through the streets of Sarajevo old town, navigating through decision points. The following was the concept of "a story guided virtual museum", implemented in the Sarajevo Survival Tools project [15]. The digital story provides the user with the historical context of the siege of Sarajevo 1992-1996, guiding him/her through the virtual museum of objects created by the citizens during that time. The virtual exhibition is divided in thematic clusters and the stories are connecting that clusters. In [16] we introduced and evaluated through user studies the concept of audio guided virtual museum. Here we implemented the audio stories to guide the visitor through the virtual collection of Bosniak Institute exhibits. The user evaluation has shown that visitors were so focused on the story that they have not noticed that movement through 3D environment was not enabled, but they could move only through clicking on hotspots in the pre-rendered images. In the computer animation of the zikr ritual in Isa bey's tekke1 [17], the animated virtual environment was exported to Unity 3D and adjusted to place the user in the middle of the animation. The user observes the dervish ritual going on around him/her and has a possibility to explore in more detail the highlighted elements. Here the main story is happening in the ritual room semahana and sub stories are connected to highlighted scene el-

ements and activated on mouse click. After the activation every sub story is implemented as a movie. The last improvement of our interactive storytelling concepts was implemented in the Isa bey's endowment project and united the interior animation of zikr ritual with the exterior virtual environment consisting of the tekke, accommodation area, soup kitchen and water mills. The main story about the endowment and sub stories about particular objects are realized in form of audio stories in corresponding areas [18]. Once the user starts the interactive environment, the main story starts; if the user is detected inside one of these activation areas, a trigger is launched to pause the main story and start the sub-story of the activated area.

3.1 Hyperlinked interactive digital storytelling

The new method we call hyperlinked interactive digital storytelling is based on interconnecting the video file of the main story with sub-stories and the interactive virtual environments using Unity hyperlinks. The method implies structuring the story scenario in such way that it consists of a main story presenting a short summary of the topic and sub-stories offering more details on particular aspects of the topic to the user. While watching the main story, at the time each of sub-stories topics is mentioned, the user can click on a link to watch the sub-story. Sub-stories have the same recursive structure of the main story, as they can also contain their sub-stories. Instead of sub-stories, interactive virtual environments (IVE) can also be linked in this hyper-structure. The general algorithm of this method is displayed in Figure 1.

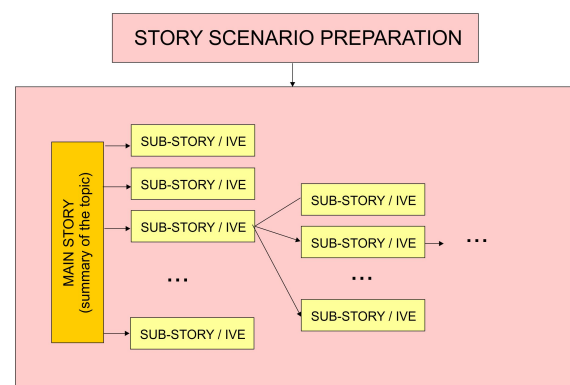


Figure 1: Algorithmic representation of the method

Through this method we aim to achieve the following contributions:

1. optimization of time the viewer spends in the application Most of the Internet users do not have time to watch a story that lasts more then 3-5 minutes. The proposed story structure offers the insight in the information content through the main story and a set

of sub-stories for viewers who have more time and/or are more interested in learning about the object. Also, the users can come back and watch sub-stories according to their time and availability.

2. adjusting the form of narrative to the concept familiar to the modern Internet era We created the story structure following the organization of HTML pages with hyperlinks. This concept is natural to the modern human media perception. If proven successful, this concept could offer an alternative form of presentation in literature and movie industry.
3. joining different media in a unique digital storyline The combination of the story about an object with an interactive 3D model of that object could enhance the user perception and immersion in the story The results of the user evaluation will show if we succeeded to reach these aims. They will show us the way we need to follow in our future interactive digital storytelling concepts development.

4 Case study the proof of concept

4.1 The Tašlihan object

The Tašlihan was the largest accommodation facility in Sarajevo during the Ottoman period. It was built between 1540 and 1543 as an endowment of Gazi Husref Bey, governor of the Bosnian province within the Ottoman Empire. It could host 20 people and 70 horses. Aside of Tašlihan was built a huge covered bazaar called Bezistan, with 52 shops. Presently there is only one wall remained of Tašlihan (Figure 2), aside of the hotel Europe garden. The Bezistan is still functional as a trade center.



Figure 2: The remains of Tašlihan, Sarajevo, Bosnia and Herzegovina

The only visualization of the Tašlihan original appearance is a part of the physical model of Sarajevo old town from XIV century exhibited in the Museum of Sarajevo (Figure 3).



Figure 3: Physical model of Tašlihan, Museum of Sarajevo

4.2 Interactive digital story

In Tašlihan application [19], the main story represents a summary of the information about the object, its history and related events and characters. It consists of 7 thematic clusters (MS 1-7). After each thematic cluster the user can activate a link to the sub-story (SS), which describes in more detail a topic mentioned in the main story. For example, in the main story the narrator says that the object was built as an endowment of Gazi Husref Bey. By clicking the link on SS1, the viewer can see the story explaining the concept of endowment in Islamic tradition. The structure of the application is presented in Figure 4. The sub-story SS1 is linked to another sub-story (SS 1-1) and the SS4 is linked to the interactive virtual model of the object. All links are displayed on the right side of

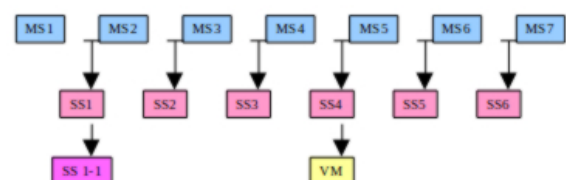


Figure 4: Structure of the interactive digital story

the window with the story player and become clickable after the predetermined time code of the main story video file (Figure 8). In [20] we showed that the storytelling is much more engaging and immersive if there is a character telling the story. Therefore in this project the story is told by Murat Bey, the first associate of Gazi Husref bey, who sponsored the construction of Tašlihan. Digital stories are created using the photos of the present appearance of the Tašlihan complex, as well as the other objects that belong to the Gazi Husref bey's endowment, such as the Begova mosque, as the character who tells the stories, Murad bey, is buried in the yard of that mosque, next to Gazi Husref bey. We also used drawings created by a digital artist and photos of the Sarajevo old town model from the Museum of Sarajevo. In stories creation we followed the principles

of film language grammar and engaged a professional narrator for voice over. 3D model of Tašlihan (Figure 5) was created in Maya and exported to Unity 3D, where textures and illumination were adjusted and optimized for online use. The geometry of the model is based on the scientific work of archaeologists and historians who excavated the remains of the object. Unity 3D has been chosen as we needed to introduce interactivity into a video. In our previous projects we used Flash, but presently it is not supported by all platforms.



Figure 5: The interactive 3D model of Tašlihan

4.3 The Unity hyperlinked story structure implementation

The screen appearance of the Tašlihan application is displayed in Figure 8. Here we briefly describe the implementation of the method. Unity 3D supports the video reproduction via MovieTexture tag. The MovieTexture can be assigned to a plane, as a common texture, and video reproduction can be controlled by three MovieTexture methods (Play, Pause and Stop). For our hyper linked stories and sub-stories we needed the current playing time, not supported by the MovieTexture, so we implemented an internal timer for each plane with the MovieTexture. The timer uses Unity Time class which stores elapsed time in seconds since start of the application. To get current video time ($T_{current}$) you have to subtract the starting time of video ($T_{start_of_video}$) and the sum of time when video was paused (T_{in_pause}) from the time since application was started ($T_{elapsed}$).

$$T_{current} = T_{elapsed} - T_{start_of_video} - T_{in_pause}$$

Explanation of the application's algorithm is given in Fig. 6. When the application is started the main story's video is played, and in the appropriate moment (time stored in the start time attribute of a sub-story's class) notification is presented and link to a sub-story is highlighted. Sub-stories are stored in the list, and sorted by their starting time. If a user clicks on the link, current story's video is stopped, its plane is deactivated (no longer visible), sub-story's plane is activated (now visible) and sub-story's video is played. If sub-story has one or many sub-stories than same process is repeated as for main story. At any moment when a user is watching sub-story he can

click the back button which takes him back to the main story. If that happens sub-story's video is paused, the sub-story's plane is deactivated (no longer visible), the main story's plane is activated (now visible) and the main story's video is played. Along with these controls a user can pause and rewind video. If the pause button is clicked the video is paused via `MovieTexture.Pause()`, the elapsed time in moment of the click is stored in timer's variable called `pauseStart` and pause button is converted to the play button. The play button resumes the video and calculates time spent in pause using earlier `pauseStart`, this time is used to calculate `Tcurrent`. If rewind button is pressed the video is stopped via `MovieTexture.Stop()`, this resets video to start, our timer is also reset and then `MovieTexture.Play()` is called. Unity scripts are used to implement the algorithm. Each plane has the script attached. The script do all actions regard plane activation/deactivation, video play/pause and timer calculation. In the script each story's data is stored in `Story` class. `Story` has attributes: name, start time, end time, link button, plane. We use these data to know exact time when to notify a user and which plane to activate. The data about each story is tuned so the transition from a current story to a sub-story is as smooth as possible. The start time of a sub-story is (manually) determined in a way that we know when context of the sub-story is mentioned in the parent story, in that moment sub-story becomes relevant, and that moment is time when sub-story should start. The start time and end time is measured in the timescale of the parent story.

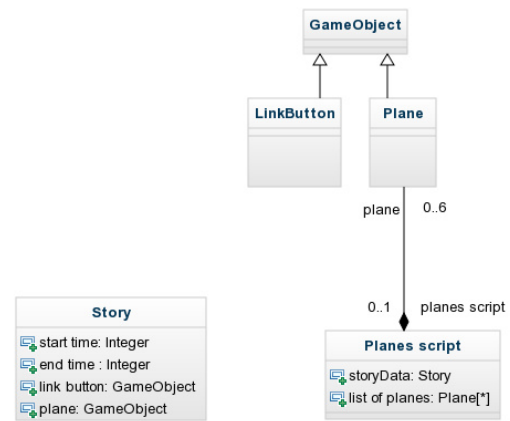


Figure 7: Class diagram

Description of each Story's [Figure 7] attribute:

- name – short Story title,
- start time – time when the story is mentioned in the parent story. We use this time to notify a user that he can click to the link of this story,
- end time – time when story stops being relevant to the parent story and time when notification is closed,

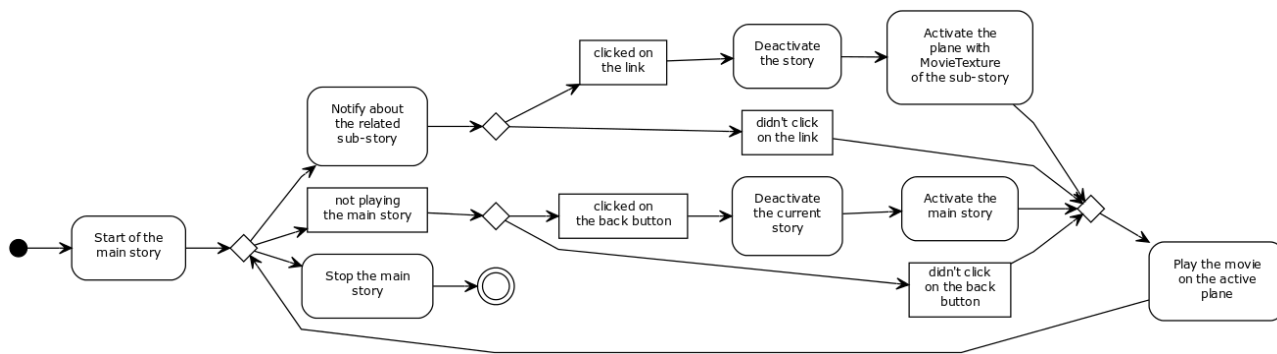


Figure 6: Activity diagram

- link button – the button which switches the parent story and the sub-story. When the story is loaded sub-stories of the current story are loaded and stored in list,
- reference to the plane with video content – we use this reference to activate the story plane.

When implementing a Unity application with video, two issues should be considered: Unity native video format is ogg theora MovieTexture has limited set of video control methods, there is no support for playing a video from random position (no seek method), no support for current playing time.

5 User evaluation

5.1 Introduction

User evaluation is the main parameter for the success of virtual cultural heritage applications. We performed two user evaluation studies: quantitative evaluation based on customer satisfaction questionnaires and qualitative evaluation based on user interviews.

5.2 Customer satisfaction questionnaires

The initial evaluation of the project was done by 9 users who filled out a customer satisfaction survey. The first group of questions was related to information perception. We have investigated what the users have learned from the application asking them questions about the notions mentioned in stories. We also evaluated the quality of interactive digital storytelling, asking the users to mark the narrative, the video and music in digital stories. The third group of questions was about the interactive 3D model of Tašlihan. Firstly, we investigated if the users got to open the model, as it is positioned as a link in one of sub-stories. Then we inquired about the quality of models geometry, textures and illumination, as well as the navigation through the model. The last group of questions considers the overall satisfaction of users, with emphasis on feeling of im-

mersion. The users could describe what they liked and disliked in the application.

The questions which contained rating of particular aspects of the project were set up according to the positive response bias [21], so the users could rate that aspect from 1-10, but in case 9 or lower was chosen they were offered to answer What would make it a "10"?

Although this initial evaluation was performed on an extremely small number of users, the results we obtained are very valuable. Most of the users rated the application and its particular aspects with 10 and felt immersion in the past of the Tašlihan object. But, more important was that they appreciated the concept of hyperlinked stories as a method to present the information on cultural heritage. One of them answered the question "What did you liked the best in the Tašlihan application? with the following statement: -"widening" of the story in a sense that the viewer who becomes interested in particular part of the story has the possibility to explore it further. Also, I have never seen something similar to this concept of mixing stories and 3D models in a unique interactive application, it's a quite new experience for me. And another one said: "this is much better than reading a tourist guide". Most of the users found as a drawback the long downloading time of the application, some of them would like to see a more detailed 3D model of the object and one of them was missing people in the virtual environment. They also mentioned that some controls over the playback of the stories should be introduced.

5.3 Qualitative data analysis - data coding

One of the most efficient methods of user evaluation is the qualitative analysis of user feedback. It is based on interviews with the users in which they express their experience during the use of the application. This kind of user evaluation is performed according to the following workflow: definition of hypotheses, interviewing the users, data coding of users feedback, analysis of coded answers and comparison with hypotheses.

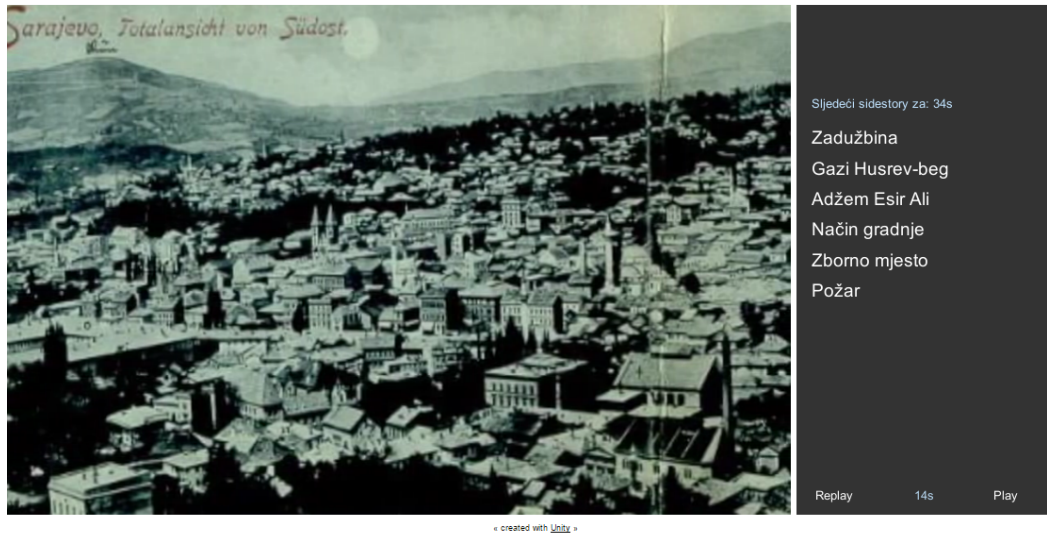


Figure 8: The Tašlihan application main story

5.3.1 Experiment design

The user experience practice has shown that 7 users will find approximately 80% of problems of an interface or application [22]. User selection included different nationalities (Spanish, Indian, Azeri, Chinese) and various academic backgrounds (Computer Science, Historians, and even some Colour Science student, etc). Five of them were experienced computer users. Average age of users was 29 years old

Question	Code	Possible value and number of answers
How interesting and engaging you found stories in the application?	Q6	Very interesting (2) Quite interesting (2) Interesting (2) Intermediate (1) Not interesting (0)
Did you see a common line between them?	Q7	Yes (7) No (0)
How immersive you found the application?	Q14	Very immersive (2) Moderately immersive (3) Slightly immersive (1) Not immersive (1)

Table 1: Illustration of data coding

5.3.2 Evaluation process and results

Qualitative data analysis is based on data coding [22]. It is a process of extracting qualitative data into quantitative form. The possible values of the qualitative data are created according to the given answers. Since participants often use different terms for the same notion or same words for different notions, it is important to perform coding as

accurate as possible, without losing too much information. The data analysis was performed in two steps: defining the hypotheses and grounding the evidence. We defined the following hypothesis:

- H1: users learn more from interactive storytelling than from linear story
- H2: interactive storytelling application makes users immersed in the past
- H3: users prefer interactive over linear storytelling

The hypotheses were generated using the constant comparison method [23]. After coding the questions (Table 1), each of them representing a particular section, we went through the data looking for patterns. At the end of the analysis we obtained the following level of hypotheses confirmation (Table 2) These results show that we still

Hypothesis	Percentage of confirmatory answers
H1	78.30%
H2	62.50%
H3	50.00%

Table 2: Hypothesis confirmation results

have to work on the interactive digital storytelling methods in order to motivate the majority of users to choose it over the linear storytelling. At the end we will quote some of the user statements we find valuable for the future research: "It supplies 3D model that allows the user to get the feeling about the building. Compared with a normal presentation, makes the user involved with the environment" "The interactive form gives freedom to select certain part of information the participant would like to listen

to again.” ”It is nice how information is displayed, the mixture of old images and 3D reconstruction. If I would need to imagine, from the current picture of the site nowadays, how it was the building before, it would not be possible for me”.

6 Conclusions

The Tašlihan application aims to present to the Internet users a cultural heritage object which does not exist any more, except the remains of one of its walls. In this presentation we used the hyperlinked interactive digital storytelling combined with the 3D virtual environment. This concept offers to the users an overview story about the history of the object, hyperlinked with sub-stories about some events and characters from its history, together with the possibility to virtually explore the reconstruction of the object. The application structure is implemented in Unity 3D. The initial user evaluation of the application shows that this concept could be appreciated by the viewers and can create the feeling of immersion. The main drawback of the Unity implementation is the long downloading time. In the future work we need to decide whether the links to sub-stories should be active all the time, or should be activated after the key notion is mentioned in the main story. We also need to introduce some controls for users to play-back the stories. The thorough user evaluation needs to be performed on a larger groups of viewers, with different professional backgrounds, ages and computer literacy.

7 Acknowledgment

The interactive 3D model of Tašlihan was created by Bojan Kerouš as a part of his graduation project at the Faculty of Electrical Engineering Sarajevo.

References

- [1] G. Davenport H. Brndmo. Creating and viewing the elastic charles. *Hypermedia Journal, Hypertext: State of the ART*, 5:43–51, 1989.
- [2] I. Smith N. Sawhney, D. Balcom. Hyperface: Narrative and aesthetic properties of hypervideo. *Hyper-text '96 Proceedings, ACM, New York*, pages 1–10, 1996.
- [3] The raptmedia editor. <http://www.raptmedia.com>. Accessed: 2015-03-01.
- [4] The madvideo tool. <http://www.themadvideo.com>. Accessed: 2015-03-01.
- [5] Open hypervideo as archive interface, interface design, 2012.
- [6] E-learning-how-tos. <http://learn.articulate.com>. Accessed: 2015-03-01.
- [7] Cacophonys player. <http://www.cacophonyjs.com>. Accessed: 2015-03-01.
- [8] Clickvid players. <http://www.clickvid.co.uk>. Accessed: 2015-03-01.
- [9] A human sanctuary project, cyprus institute. <http://public.cyi.ac.cy/scrollsDemo>. Accessed: 2015-03-01.
- [10] S. Rizvić, D. Pletinckx, S. Pescarin. Keys to Rome The Next Generation Virtual Museum Event. South East European Digitization Conference SEEDI 2014, Belgrade, Serbia, 2014.
- [11] C. Miller. Digital Storytelling. *Elsevier*, 2008.
- [12] A. Glassner. *Interactive Storytelling*. A. K. Peters, 2004.
- [13] L. Manovich. *The Language of New Media*. 1st MIT Press pbk. ed, Cambridge, Mass, 2002.
- [14] N. Kraljic. Interactive video virtual tours. In *Proceedings of CESCOG*, 2008.
- [15] S. Rizvić, A. Sadžak, et al. Interactive Digital Storytelling in the Sarajevo Survival Tools Virtual Environment. *SCCG*, 2012.
- [16] Sanda Šljivo. Audio Guided Virtual Museums, Central European Seminar on Computer Graphics, Bratislava, Slovakia. 2014.
- [17] Merisa Huseinović, Razija Turčinodžić. Interactive animated storytelling in presenting intangible cultural heritage. 2013.
- [18] S. Rizvic, A. Sadzak, et al. *Interactive Storytelling about Isa Bey Endowment, Review of the National Center for Digitization*. Faculty of Mathematics, Belgrade, 2014.
- [19] Taslihan interactive storytelling application prototype. <http://h.etf.unsa.ba/han/sd/Build.html>. Accessed: 2015-03-01.
- [20] S. Rizvic, A. Ferko, A. Sadzak, et al. *A Piece of Peace in sWARajevo - Locally and Globally Interesting Stories for Virtual Museums*. In *Proceedings of the Digital Heritage International Congress*, 2013.
- [21] W. Kamakura V. Mittal. "Satisfaction, Repurchase Intent, and Repurchase Behavior: Investigating the Moderating Effect of Customer Characteristics". *Journal of Marketing Research*, 2001.
- [22] C.B. Seaman. Qualitative methods in empirical studies of software engineering. 25(4), 1999.

- [23] Feng J.H. Hochheiser-H. Lazar J. *Research Methods In Human-Computer Interaction*. Wiley, 2010.

CellUnity - an Interactive Tool for Illustrative Visualization of Molecular Reactions

Daniel Gehrer*

Supervised by: Mathieu Le Muzic[†], Ivan Viola[‡]

Institute of Computer Graphics and Algorithms
University of Technology
Vienna / Austria

Abstract

CellUnity is a tool for interactive visualization of molecular reactions using the Unity game engine. Current mesoscale visualizations commonly utilize the results of particle-based simulations, which account for spatial information of each single particle and are supposed to mimic a realistic behavior of the metabolites. However, this approach employs stochastic simulation methods which do not offer any control over the visualized output. CellUnity, on the other hand, exploits the results of deterministic simulations which are purely quantitative and in that way offering full user control over the spatial locations of the reactions in the visualization. The user is able to trigger reactions on demand instead of having to wait or search for a specific type of reaction event, while the quantities of displayed molecules would still be in accordance with real scientific data. CellUnity exploits the simulation results in real time and allows the user to freely modify simulation parameters while the system is running. The tool was realized in Unity, a cross-platform game engine that also comprises a free version with adequate functionality and therefore enables easy deployment of the project.

Keywords: Unity, visualization, molecular reactions, quantitative simulation, interactive visualization

1 Introduction

Biochemistry allows a deep insight into cells and the synergy of molecular processes. Without any visual explanation, biochemistry can be difficult to understand [1]. Hence, it is necessary to visualize these processes to gain a better and more intuitive understanding of what is happening inside a cell [2]. For learning and comprehension purposes it is also important to provide an interactive, game-like environment in order that students can immediately experience the impact of modifications in a cellular envi-

ronment [3]. Scientific illustrators usually utilize animated storytelling principles to visually explain molecular activities, e.g. a metabolic pathway. To achieve this, corresponding particles and reaction events have to be shown in a story-structured manner [1]. Available mesoscale visualization tools commonly utilize the results of particle-based simulations to generate illustrations depicting reactions of a given biochemical process. Particle-based simulations determine spatial information of each single particle and are supposed to mimic a realistic behavior of the metabolites. However, this approach does not offer any control over the visualized output [1]. In particle-based simulations it is extremely difficult to track a specific particle due to the chaotic motion. Also reactions cannot easily be observed in the complex environment [2]. Due to this problem, it is challenging to comprehend reactions describing a biochemical process. Even when single particles are tracked and brought to focus, there is still no guarantee that a desired or an interesting event will happen [1].

The goal of this project is to create a tool for interactive visualization of an illustrative molecular environment. The functionality and the implementation is inspired by the paper of Le Muzic et al. [1]. CellUnity exploits the results of deterministic simulations which are purely quantitative. This offers, in contrast to the existing approaches, full user control over the spatial locations of the reactions and avoids the chaotic diffusion motion [1]. The user is able to trigger reactions on demand instead of having to wait or search for a specific type of reaction event, while the quantities of the displayed molecules would still be in accordance with real scientific data. This makes it possible for the user to follow a specific reaction chain in a realistic environment, which is greatly valuable for the user's comprehension and for illustration purposes. Parameters such as reaction rates and particle quantities can be changed while the system is running. The impacts of these changes are immediately visualized.

Often, existing visual simulation environments like Zig-Cell3D are implemented as proprietary research prototypes that cannot be freely deployed on any machine [4]. Other tools like Molecular Maya or BioBlender are great for visualization but the created environments cannot be

*daniel.gehrer@student.tuwien.ac.at

[†]mathieu.muzic@tuwien.ac.at

[‡]ivan.viola@tuwien.ac.at

animated using a simulator [5][6]. The main contribution of this work is the implementation of such a visualization and simulation tool in Unity to enable easy and free deployment. Unity is a cross-platform game engine that also comprises a free version with adequate functionality [7]. CellUnity provides a user interface to create simple bio-molecular environments. It is possible to import molecular structures available from public databases, define molecule quantities, reaction rates, and even to locate individual particles in the environment. The settings can also be exported to bioinformatics standard formats for the usage in external applications.

2 State of the Art

There are already tools for visualizing molecular reactions depicting a biological pathway. All of them have been designed for a slightly different purpose. Yet they all share the goal to provide insight into biological processes by visualizing a cellular environment at mesoscale levels. In this chapter various available tools are examined in respect to their visualization capabilities for scientific correct mesoscopic storytelling to explain cellular activities.

2.1 Tools for Molecular Visualization only

Some tools like Molecular Maya, BioBlender and ePMV focus solely on visualizing cellular environments [5][6][8]. These tools are all implemented as plugins for 3D computer graphics software and use the host application for visualization. They are all capable of importing molecules from the RCSB Protein Data Bank and they are all available for free [9]. Molecular Maya uses Maya as host application, BioBlender uses Blender and ePMV can be used with Blender, Cinema4D and Maya [5][10][6][11][8][12].

Molecular Maya supports various representation forms and also enables the user to easily extend structures, for example creating surface meshes and biological units [5]. BioBlender can animate transitions of conformations and visualize various molecular features, e.g. the electrostatic potential (EP) and the molecular lipophilicity potential (MLP). This kind of representation makes the nano scale world more understandable and is making it easier to conceive invisible phenomena such as hydrophathy or charge [6]. The embedded Python Molecular Viewer (ePMV) does not only import molecules from different file formats but also keeps the link between structure file and the model. That way changes that are applied to the structure file after the import, are also applied to the model. The generated model is not just a static structure but can also be manipulated by the 3D host program or by python scripts that interact with ePMV.

All mentioned tools can be used to illustrate molecules but the models do not convey information about its function. To illustrate a cellular environment, the illustrator has

to model the molecular processes manually using the host applications default tool set, which is a time-consuming and expensive task, taking up to months or years [1].

2.2 Tools for Molecular Visualization and Simulation

2.2.1 Visualization of Signal Transduction Processes

Falk et al. developed a visualization framework to explore simulation data of a virtual cellular environment [2]. The goal of the work was to highlight events of interest in the confusing environment. It especially helps Biologists to follow signaling molecules through the cell. The user can interactively select individual molecules and zoom into the virtual cell. It is possible to visualize individual molecules, their tracks, or reactions. The work is suitable for detailed, realistic, spatial simulations, where each molecule is an independent agent. A simulation usually covers several hundred frames. The user can step through each frame by keystrokes. The work also includes a virtual microscope to create images which can be compared with results from wet lab experiments. While the analyzing tool is interactive, the simulation is not [2]. The user has to perform the simulation again before it is possible to see the effects of the changes made. Also the tool is not openly available and therefore only used by a small set of users [2].

2.2.2 MCell and CellBlender

MCell (Monte Carlo Cell) is referring itself as micro physiological simulator [13]. It is a program to simulate the movements and reactions of molecules within and between cellular regions. For simulation, MCell uses spatially realistic 3D models and specialized Monte Carlo algorithms. It is intended to realize as realistic simulations as possible. The model can contain multiple compartments, which represent enclosed parts, e.g. organelles [13]. The meshes can be obtained from segmented volumetric imaging data or from CAD (computer-aided design) software [14]. MCell is free of charge and available for Linux, OS X, and Windows [13]. The model and the simulation conditions are defined in modular, human-readable text files, using a model description language [14].

CellBlender is an add-on for the free and open-source 3D computer graphics software Blender [11] [15]. The add-on is closely linked to MCell and enables the user to perform integral modeling tasks in Blender. It is possible to create, edit and visualize cellular models for the use in MCell. The simulation results generated by MCell, again, can be visualized in Blender, including the locations and states of participating meshes and molecules [15].

2.2.3 ZigCell3D

ZigCell3D is a software for modeling, simulating and visualizing an entire cell. The visualization covers several

orders of magnitudes, the full range from the cell surface to organelles and molecules down to the atomic level. The system also includes a virtual fluorescence microscope. For simulation two different approaches are used. On the one hand particle-based Brownian dynamics simulation, and on the other hand simulations based on Reaction Diffusion Master Equations (RDME), which have less spatial resolution but better performance [4].

ZigCell3D provides a real time interactive environment, where model parameters can be changed and the resulting effect can be analyzed in the 3D visualization of the cell. The user can select particles and can analyze the underlying rules and mathematical expressions that are responsible for the creation of the selected component or molecule. That frees the user from guessing the possible origin and bridges the divide between quantitative sciences and math-free wet-lab biology [4].

3 Methods

In CellUnity, the user has to set up the molecular environment before it can be simulated. The setup consists of importing different molecule species, setting the initial quantities, defining the reaction equations and setting the compartment size. The tool is interactive, so that the user is able to explore and immerse into a virtual 3D cellular environment. It is possible to track molecular compounds and also trigger reactions manually, while respecting quantities obtained via scientific simulation. In this chapter, the concepts and tools used in CellUnity are introduced. In the next chapter the concrete implementation is explained.

3.1 Development Environment

The project is built in Unity, a cross-platform game engine. Unity is not only a game engine but also includes an integrated development environment (IDE). Unity was chosen as framework because it is easy to use, quick to learn and there is also a free version with adequate functionality to realize this tool. Moreover, this enables the project to be easily shared and deployed, and allows the user to modify or extend the project with little effort. Additionally, Unity provides built-in methods for visualizing 3D objects and has a built-in physics engine. These features speed up development and avoid that the project has to be created from scratch. Furthermore, the Unity editor can be extended easily to include custom plugins, which seamlessly integrate into the Unity interface [7].

3.2 Visualization of Molecules

Molecules are visualized at atomic resolution. CellUnity can import molecule species from the file system using PDB files or can download the structure information automatically from the PDB webserver using a given PDB ID

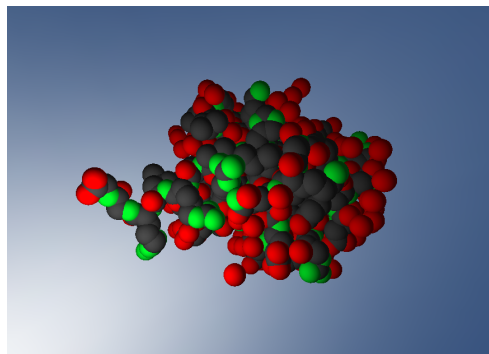


Figure 1: Visualization of an ubiquitin [16] molecule in CellUnity

[9]. The representation of a molecule is automatically created using the atom definitions in that PDB file. The imported molecules are displayed as bunch of spheres, each representing an atom. The locations of the atoms are in accordance with the PDB file. The size of each atom corresponds with the Van der Waals radius of the associated element. This representation is often referred as Van der Waals representation [1]. An example is shown in figure 1.

3.3 Simulation

CellUnity is coupled to a simulator to mimic a realistic behavior in the visualization. For simulation the biochemical simulator COPASI is used [17].

As soon as the simulation is started, the user defined initial state is transmitted to the simulator. Since CellUnity only needs the number of reactions occurred, the simulation is purely quantitative. The simulation is performed step by step. After each step, the results are transmitted back to CellUnity and the reactions are performed in the visualization. It is also possible to modify simulation parameters after each step. The duration of such a step is adjustable by the user.

Reaction events are solely triggered by an omniscient intelligence (OI), like proposed by Le Muzic et al. [1]. In this system, molecules are passive agents, according to the definition by Kubera et al. [18]. Unlike in spatial-based simulation methods, molecules in CellUnity are unable to initiate reactions but can only receive reaction orders from the OI. The OI is influenced by the simulator and controls the molecules accordingly. Thus reactions in CellUnity do not just happen but are actively forced. The OI uses the current simulation state and takes action to achieve the same state in the visualization. Concretely, the OI reads out the quantity of reactions that occurred in the simulation for each reaction type, and forces the same quantity of reactions to happen in the visualization. Therefore the simulation and the visualization are quantitatively synchronized [1].

When a reaction is initiated, the OI selects random or user selected candidates according to the species of the

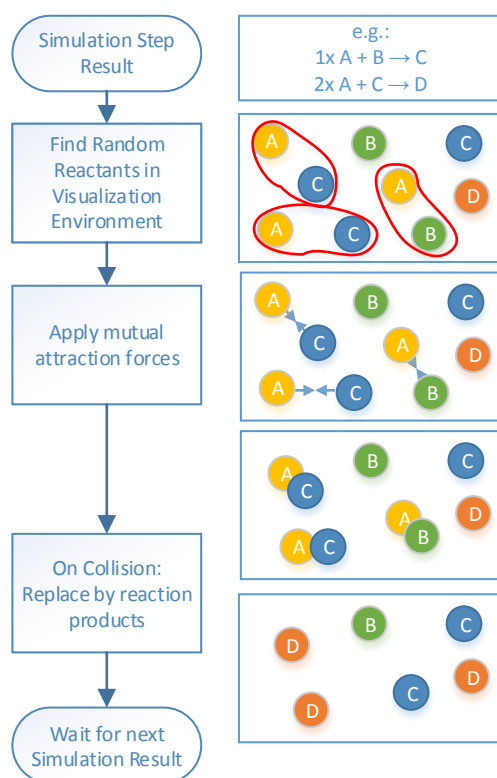


Figure 2: Example: Apply simulation results in visualization.

reactants and applies mutual attraction forces to force reaction partners to meet. As soon as they collide, the reactants are replaced by the reaction products. An example is shown in figure 2. The motion of molecules and also the collision detection is realized using Unity's built-in physics engine. Depending on if colliding molecules should react, the reaction takes place, if not, they repel each other via bouncing motion [1].

3.4 Navigation and Storytelling

To allow the user to navigate through the molecular environment, a navigation similar to a first-person-game is enabled. The user can turn around using the mouse and move with the W, A, S, D keys. Clicking on a molecule selects it. The user can adjust the view to automatically follow the selected molecule in the space or tag it for a reaction. If a molecule is tagged for a reaction, its priority will be set to react first when the OI initiates a new reaction. This allows the user to easily follow a reaction chain along a metabolic pathway. This is an easy way to comprehend reaction chains without having to wait until reactions happen on themselves, and is useful for storytelling.

3.5 Data Persistence

Unity usually serializes game data into so-called assets to persistently save them [7]. CellUnity uses this feature to

save environmental data like molecule species and reactions. Molecules are implemented as GameObjects and therefore can be saved and restored in scenes when Unity is in edit mode. The position of every molecule is also preserved that way. In game mode the scene cannot be saved but the current state can be exported to an SBML file. The export functionality is available in edit mode as well. The SBML functionality is acquired by an external library.

4 Implementation

CellUnity is implemented as a project inside Unity. Custom editors are used to allow the user to configure the molecular environment. CellUnity's implementation is divided into individual classes. It is heeded that responsibilities of each class is well defined, to ensure coherent program modules that are as independent as possible. The CellUnity Environment (CUE) implements the model of the cell, the custom editor serves as controller for this model and Unity provides the visualization. Together these components form a model-view-controller.

One custom editor window is implemented for CellUnity. Via this editor the user can model and modify the environment. It is possible to import molecule species from PDB, to add and remove reactions from the system, configure simulation properties and export the environment to an SBML file. The target of the changes made in the editor is the CUE, which holds all the environment properties and definitions.

4.1 CellUnity Environment

The CellUnity Environment (CUE) is the class that holds all environmental properties and definitions. The entire system can only contain one instance of a CUE, therefore it is implemented as singleton. The CUE contains the defined molecule species and reactions, the volume of the compartment, a molecule manager, a reaction manager and a simulation manager. Each manager focuses on a separate task. They are described in detail in later sections.

4.2 Saving

Because all the environmental information is stored in the CUE, it makes sense to simply serialize the instance to persistently save the entire model when Unity is closed. Unity already provides automatic serialization methods. However, a few specific characteristics must be considered when used. Multiple references to one instance of a class are serialized multiple times, therefore, for every reference a new instance is created after restoring. This behavior is not satisfactory for species and reaction instances. Therefore these classes are derived from ScriptableObject. ScriptableObjects are serialized only once and multiple references are restored correctly [7].

4.3 Compartment

The CUE currently only supports one compartment and it has to be in the shape of a sphere. To keep the molecules inside the compartment, collisions with the compartment wall must be detected and the particles must bounce off. The physics engine of Unity does not support inverted colliders, which would be required. Therefore the desired behavior must be implemented by user code [7]. For each molecule, the distance to the compartment center is calculated. If the distance exceeds the radius, the distance is set to the value of the radius and the velocity of the molecule is inverted.

4.4 Molecules

In CellUnity, molecule representations are GameObjects with a Molecule-script attached. The Molecule-script applies molecular behavior to the GameObject. Each molecule is an instance of a specific molecule species. To make it easy to insert new molecules, each species has a prefab asset. A prefab is a Unity asset type that allows to store a GameObject with all components and properties. It acts as a template from which it is possible to create new instances in the environment [7]. The prefab is automatically created when a new species is imported.

4.5 PDB Import

CellUnity can either import PDB files from the file system or download them directly from the PDB website. PDB files are text files which contain 3D structure information of biological macromolecules [19]. For the molecule representation, only the atom positions in the file are considered. To gather this information, a simple PDB parser was written.

When a new molecule species is imported, at first, a new MoleculeSpecies-instance is created and added to the CUE. Then an empty GameObject is created, which serves as the main object of the molecule. The main object gets the Molecule-script attached and the newly created species-instance assigned. All atoms defined in the PDB file are now created as sphere-primitives and are added as sub-objects to the main object. To get a Van der Waals representation of the molecule, the size, location and color of each sphere is set accordingly. For performance reasons only one spherical collider that is considered by the physics engine is used for the whole molecule. The newly created molecule is then saved to a new prefab asset and assigned to the new species as the template for the molecules.

4.6 Molecule Manager

The assignment of the MoleculeManager is to keep track of all molecules in the system. When the play mode in Unity is activated, each molecule registers itself to

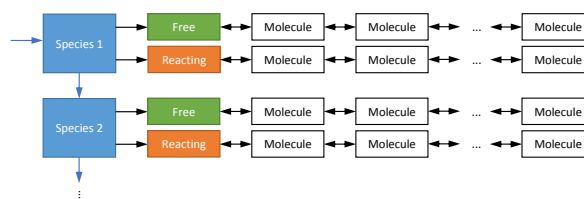


Figure 3: Schematic Diagram of the Molecule Manager

the MoleculeManager of the CUE. In the manager, all molecules are organized in separate lists, depending on their species and whether they are free or already in use for a reaction. These lists are implemented as double linked lists. One molecule can only be in one list at a time. This enables to find free molecules quickly and efficient, which is important for the preparation of reactions. The organization of the molecule manager is depicted in figure 3.

When a new reaction is initiated, the reaction manager asks the molecule manager to find a set of molecules of specific species that are near together. Due to the organization in the molecule manager, the nearest free molecule of a specific species can be found in $O(n + m)$ where n is the number of species, which is usually very small, and m is the number of molecules which are “free” and of the defined species, meaning only a fraction of all molecules. When a molecule’s state changes from “free” to “reacting”, it has to be removed from the “free” list and added to the “reacting” list. This state change can be performed in $O(1)$.

4.7 Automatic Molecule Placement

Since it is impractical to place each molecule manually, CellUnity offers to place a defined initial quantity of molecules automatically and randomly. The initial quantity can be set in the species editor. A problem that can possibly occur is that due to the randomness two or more molecules are placed too near together so that their colliders intersect. When this happens in Unity, particles repel each other with an unusually strong collision response which we ought to avoid. In CellUnity the problem is solved using the physics engine itself. The initial drag of the molecules is set to a very high value. As a result, colliding molecules repel each other gently until they do not intersect each other any longer and then remain steady next to each other. This procedure is only performed once, when the molecules are placed.

4.8 Reactions

Reactions that are possible in the system must be defined to the CUE. They are defined as ReactionType-instances. A reaction type consists of one or more reactants, zero or more products and a reaction rate. Reactants are the molecule species that are needed to perform a reaction.

Products are molecule species that are produced when a reaction was performed. The rate defines how many reactions should happen in a given amount of time. The reaction law is by default “Mass action (irreversible)” and cannot be changed in the current version of CellUnity.

4.9 Reaction Manager

The assignment of the ReactionManager is to initiate reactions and to perform them when all reactants collide. Reactions are usually initiated by the simulation manager. When a reaction of a specific reaction type is started, a new ReactionPrep (short for reaction preparation) object is created, which stores all important information about the reaction. Then the reaction manager asks the molecule manager to choose some free molecules of the species of the reactants for the reaction. The user can influence which molecules are chosen next by selecting them. If no molecule is selected, the reactants are picked randomly. If not enough molecules are available, the reaction is noted and delayed until enough molecules are available. This is important to guarantee correct molecule quantities on the long run. Such a delay can happen when the visualization is slower than the simulation, for example when the reacting molecules are located far apart, or when they hit obstacles which slow them down before reacting. However, if enough molecules are available, they are linked with the ReactionPrep object. Every molecule can only be linked to one ReactionPrep object at a time. Molecules linked to the same ReactionPrep instance and therefore are reactants of the same reaction, attract each other. This ensures that they will collide. When two molecules collide, they inform the reaction manager. The reaction manager checks if they belong to the same reaction, if yes, they are both tagged as “ready”. As soon as all reactants are ready, the reaction is performed. The reactants are then replaced by new product molecules.

When a reaction is initiated, the reactants attract each other and accelerate towards each other. Due to their physical properties they possibly do not collide immediately but start to orbit the common barycenter. This can result in an endless circulation with the molecules never collide. To avoid this, a drag is set in the environment for all molecules. As a consequence orbiting molecules slow down and collide after a short time.

4.10 Simulation

CellUnity is coupled with COPASI, a tool for quantitative modeling and simulation. COPASI is used to simulate the user defined molecular environment. The communication is enabled via the C# application programming interface (API) provided by COPASI [17]. The simulation is started and administered by the simulation manager. The manager is also responsible for the data transfer with the simulator as well as the utilization of the simulation results.

4.11 Simulation Manager

Prior to the real-time simulation of the environment, the CellUnity model has to be transferred to COPASI. The compartment, the species and the reactions from the CUE are added to COPASI via the API. The initial quantities of the species in COPASI are set to the count of the species currently located in the CUE. When the model is changed, it is re-transferred to COPASI. Because CellUnity only pursues of quantitative correctness, everything needed from the simulator are the number and types of reactions performed in the simulation. To gather this value for each reaction type, a “global quantity” model value is added. The value is defined as the ParticleFlux of the particular reaction. The type of the model value is set to “ode”, so the value is the total value of performed reactions of this type.

The simulation is performed in steps. In CellUnity, there is a time for the “visualization step” and a time for the “simulation step” that the user can define. The “visualization step” is the real time interval of a step. The “simulation step” is the time simulated in such a step. After each simulation step, the ParticleFlux of each reaction is compared with the value before that step. The difference is the number of reactions performed during this step. The same number of reactions is then initiated in the visualization.

5 Summary

This paper presents an interactive tool for illustrative visualization of molecular reactions. It enables the user to build a simple molecular environment and simulate it in real time. It is possible to import molecular structures available from public databases, define reactions, and locate molecules in a compartment. Existing visualization tools commonly utilize particle-based simulations to generate illustrations depicting reactions. This approach provides highly realistic visualization, however, it does not offer any control about the visual output. Due to this, it can be hard to follow a specific chain of reactions, because reactions occur randomly and it is not guaranteed that anything interesting will happen in the user’s sight. CellUnity, on the other hand, allows the user to trigger reactions and can automatically follow molecular reactions along a metabolic pathway. Even though the user interacts with the environment, the visualization remains in accordance with real scientific data. This enables the user to experience and comprehend metabolic processes. The model created in CellUnity can be exported as SBML file and used in other applications. Another advantage is that only free software is used to develop CellUnity. Hence, CellUnity can be easily deployed, modified and extended by everyone.

References

- [1] Mathieu Le Muzic, Julius Parulek, Anne-Kristin Stavrum, and Ivan Viola. Illustrative visualization of molecular reactions using omniscient intelligence and passive agents. *Computer Graphics Forum*, 33(3):141–150, June 2014.
- [2] Martin Falk, Michael Klann, Matthias Reuss, and Thomas Ertl. Visualization of signal transduction processes in the crowded environment of the cell. In *Proceedings of the 2009 IEEE Pacific Visualization Symposium*, PACIFICVIS '09, pages 169–176, Washington, DC, USA, 2009. IEEE Computer Society.
- [3] Marc Prensky. Computer games and learning: Digital game-based learning. *Handbook of computer game studies*, 18:97–122, 2005.
- [4] P. de Heras Ciechomski, M. Klann, R. Mange, and H. Koepl. From biochemical reaction networks to 3d dynamics in the cell: The zigcell3d modeling, simulation and visualisation framework. In *Biological Data Visualization (BioVis), 2013 IEEE Symposium on*, pages 41–48, Oct 2013.
- [5] Molecular Maya Toolkit website. <http://www.molecularmovies.com/toolkit/>. Accessed: 2014-08-21.
- [6] Raluca Mihaela Andrei, Marco Callieri, Maria Francesca Zini, Tiziana Loni, Giuseppe Maraziti, Mike Chen Pan, and Monica Zoppè. Bioblender: A software for intuitive representation of surface properties of biomolecules. *CoRR*, 2010.
- [7] Unity Technologies. <http://www.unity3d.com/>. Accessed: 2014-08-21.
- [8] Graham T. Johnson, Ludovic Autin, David S. Goodsell, Michel F. Sanner, and Arthur J. Olson. epmv embeds molecular modeling into professional animation software environments. *Structure*, 19(3):293–303, 2014.
- [9] Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The protein data bank. *Nucleic Acids Research*, 28(1):235–242, 2000.
- [10] Autodesk Maya. <http://www.autodesk.de/products/maya/>. Accessed: 2014-08-21.
- [11] Blender Online Community. Blender - a 3d modelling and rendering package, <http://www.blender.org>. Accessed: 2014-08-21.
- [12] Maxon Cinema4D. <http://www.maxon.net/de/products/cinema-4d-studio.html>. Accessed: 2014-08-21.
- [13] MCell website. <http://mcell.org/>. Accessed: 2014-08-21.
- [14] Rex A. Kerr, Thomas M. Bartol, Boris Kamubsky, Markus Dittrich, Jenchien Jack Chang, Scott B. Baden, Terrence J. Sejnowski, and Joel R. Stiles. Fast monte carlo simulation methods for biological reaction-diffusion systems in solution and on surfaces. *Nucleic Acids Research*, 2008.
- [15] CellBlender website. <https://code.google.com/p/cellblender/>. Accessed: 2014-08-21.
- [16] PDB ID: 1UBI Ramage, R. and Green, J. and Muir, T.W. and Ogunjobi, O.M. and Love, S. and Shaw, K. Synthetic, structural and biological studies of the ubiquitin system: the total chemical synthesis of ubiquitin.
- [17] Stefan Hoops, Sven Sahle, Ralph Gauges, Christine Lee, Jrgen Pahle, Natalia Simus, Mudita Singhal, Liang Xu, Pedro Mendes, and Ursula Kummer. Copasia complex pathway simulator. *Bioinformatics*, 22(24):3067–3074, 2006.
- [18] Yoann Kubera, Philippe Mathieu, and Sébastien Picault. Everything can be agent! In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, pages 1547–1548, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.
- [19] wwPDB. Protein data bank contents guide: Atomic coordinate entry format description version 3.30. ftp://ftp.wwpdb.org/pub/pdb/doc/format_descriptions/Format_v33_A4.pdf.

Real-Time Rendering & Illumination

Fast Photon Gathering in Progressive Photon Mapping Using GPGPU

Tomaáš Lysek*

Supervised by: Pavel Zemčík†

Department of Computer Graphics and Multimedia
Faculty of Information Technology, Brno University of Technology
Brno / Czech Republic

Abstract

This paper describes an experimental method implementing Progressive photon mapping on contemporary hardware - PC with GPU. The overview of Progressive photon mapping as well as its GPGPU-specific modification. The experimental results are shown as well along with the performance measurements.

Keywords: Global illumination, photon mapping, gpgpu, opengl

1 Introduction

From the beginning of computer graphics, there was an effort to make nice photorealistic images. One way to do this is to use global illumination methods. These methods have high computational cost and they are not always suitable for massive parallelism on graphics acceleration hardware, such as GPU. Photon mapping is a relatively new approximation of global illumination and Progressive photon mapping is its very promising variant. This paper proposes a naive implementation of Progressive photon mapping on contemporary GPU and proposes acceleration approaches on this naive solution.

2 Related work

Best photorealistic rendering results today are achieved using methods belonging to the global illumination ones. These methods attempt to simulate physically correct light propagation in scenes, and the basis for all of the global illumination methods was set in 1986 when James T. Kajiya [5] formulated the rendering equation.

Rendering equation describes how to precisely compute reflected radiance in a certain point by summing light contribution from all directions in hemisphere around examined point. Using this approach, it is possible to compute precise light propagation in the scene. The major

problem of this approach is that this computation is done by computing integral through hemisphere and computing this integral is nearly computationally impossible for large scenes.

In the paper that introduced the rendering equation, the path tracing method was proposed as well. Path tracing samples Rendering equation by examination of random direction in hemispheres. To get good result, very many random paths have to be computed. Path tracing is, in fact, using Monte Carlo approach for integral computation and thus the methods based on path tracing are called Monte Carlo methods.

Path tracing was probably the first complete global illumination technique (although radiosity was invented earlier, it assumes only diffuse light propagation) and using this technique, it was possible to compute nice photorealistic images. Extension of path tracing called bidirectional path tracing was invented independently in 1993 [6] and 1994 [7]. Bidirectional path tracing traces paths from eye and from light simultaneously and from these two paths, it computes illumination. It is possible (in scenes with a lot of indirect illumination) to compute photorealistic images in lower time using bidirectional path tracing comparing to the original path tracing. Another extension of path tracing is so called Metropolis light transport [8] and this technique sets another examined path from previous path by mutation of such path.

To achieve good photorealistic results using Monte Carlo raytracing methods, many paths per pixels need to be examined and it is very time consuming. Another way how to approximate rendering equation is using Photon mapping. Photon mapping was invented by Henrik Wann Jensen in 1996 [4]. It is two-pass algorithm; in the first pass light contribution in scene is computed by sampling light distribution from scene, sample of light is called photon and the photons are saved in photon maps covering surfaces of the scene. In the second pass, an extended raytracing is used to compute final image. When this extended raytracing computes local illumination model (for example by Phong lighting), illumination from photon maps is included.

For good results in Photon mapping, large number of

*xlysek03@stud.fit.vutbr.cz

†zemcik@fit.vutbr.cz

photons has to be saved in photon map and if photon map is really big, searching in photon map becomes slow. Extension of normal photon mapping, called Progressive photon mapping [2] addresses this problem. The difference from the standard Photon mapping is that Progressive photon mapping computes many smaller photon maps and progressively improves results from another photon maps. In 2009, Stochastic progressive [1] photon mapping was proposed, extending Progressive photon mapping by distributed raytracing effects, such as depth of field, motion blur, etc.

All of the above described global illumination methods are consistent - this means that with increasing rendering time method is approaching correct result. Monte Carlo raytracing methods are unbiased - this means that even if we compute one path per pixel and average large amount of this images, we still get correct result. On the other hand Photon mapping methods are biased and if we average many of rendered images we do not generally get correct result.

3 Progressive photon mapping

Radiance estimation in photon mapping is an approximation of Rendering equation. Luminance at point x heading in direction $\vec{\omega}$ is computed as:

$$L_r(x, \vec{\omega}) \approx \frac{1}{\pi r^2} \sum_{p=1}^N f(x, \vec{\omega}, \vec{\omega}_p) \Delta \Phi_p(x, \vec{\omega}_p) \quad (1)$$

where $\Delta \Phi_p(x, \vec{\omega}_p)$ is photons flux saved in photon map. $f(x, \vec{\omega}, \vec{\omega}_p)$ is bidirectional reflectance distribution function. Sum involves N nearest photons from point x in photon map. Nearest photon in space generates sphere and because it is possible to assume that photons are accumulated from flat surface, the result is divided by area of circle where r is distance of farthest photons from point x .

For removing low-frequency noise in result images, photon map has to have many photons, possibly infinitely many. If photon map has infinite number of photons and in radiance estimation is gets fraction of this infinite photons, radiance is estimated in radius approaching zero at the limit. From this observation, it is possible to assume that best results are obtained by radius as small as possible. With increasing number of photons in the photon map, both memory and time complexities are increasing.

There was an effort to divide final large photon map into several smallest photon maps, compute some sort of data and get better result in faster time. One way was averaging lot of computed images, but this does not lead to consistent result.

Another way was invented with Progressive photon mapping. This multiple-pass technique reorders photon mapping in proper way and ensures that with another passes, consistent result is obtained.

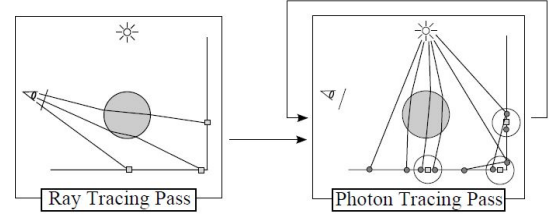


Figure 1: Progressive photon mapping schema[2]

Scheme of Progressive photon mapping is shown in Figure 1. First, raytracing is performed and after this, photon tracing passes are performed. Raytracing is performed for saving special positions in scene - so called hitpoints. Hitpoints are saved when ray intersect with diffuse surface, and in each hitpoint lot of needed data are saved. This data contains: position, material, normal, pixel location, pixel weight, current photon radius, accumulated photon count and accumulated reflected flux. Using this data it is possible to synthesize final image.

Photon tracing pass is divided into iterations, with fixed (smaller than in normal photon mapping) number of photons. Theoretically it is possible to process infinite number of photons on limited memory. In each iteration, new random photon map is created and then for each hitpoint, illumination from photon map is computed. As it was described above, radius decreases with each iteration. Equation for computation new radius is written as:

$$\hat{R}(x) = R(x) \sqrt{\frac{N(x) + \alpha M(x)}{N(x) + M(x)}} \quad (2)$$

where $R(x)$ respectively $\hat{R}(x)$ is radius in hitpoint x respectively new radius in hitpoint x . $N(x)$ is number of photons saved in hitpoint x , $M(x)$ is number of new photons in current iteration in current radius $R(x)$. α is value in range 0 - 1 and indicate how much new photons will be added to illumination and how fast radius will be decreasing.

New flux $\tau_{\hat{N}}(x, \vec{\omega})$ in hitpoint x is computed as:

$$\tau_{\hat{N}}(x, \vec{\omega}) = \tau_{N+M}(x, \vec{\omega}) \frac{N(x) + \alpha M(x)}{N(x) + M(x)} \quad (3)$$

where $\tau_{N+M}(x, \vec{\omega})$ is sum of flux from previous iteration and current iteration computed by equation 1 and values $N(x)$, $M(x)$ and α is same values as in equation 2.

Final luminance in point x heading in direction $\vec{\omega}$ is computed as:

$$L(x, \vec{\omega}) \approx \frac{1}{\pi \hat{R}(x)^2} \frac{\tau_{\hat{N}}(x, \vec{\omega})}{N_{emitted}} \quad (4)$$

It is possible to render image after each photon tracing iteration or after all iteration has been proceed.

Using progressive photon mapping it is possible to render whole global illumination and very similar images like using monte carlo raytracing methods. Progressive photon mapping excel in rendering specular diffuse specular path (SDS path) above all other monte carlo raytracing methods [2]. SDS path means that path from light is connected through any number of specular surfaces, reflected on diffuse surface and then again reflected through any number of specular surfaces to eye. This effect can be seen for example on bottom of swimming pool.

4 GPGPU model

Graphics Processing Units have a unique ability to accelerate general purpose tasks. Graphics cards were made in order to accelerate realtime computer graphics; however, lately the GPUs have been able to compute nearly any kind of programmable task.

Few programming languages exist for programming GPUs, the most used being CUDA and OpenCL. They vary by notation and capabilities and because this paper is using OpenCL for execution, OpenCL notions will be presented.

Execution model

Graphics card consist of a set of streaming multiprocessors. Streaming multiprocessors can be viewed as big enhanced SIMD (single instruction single data) processors. One thread of parallel computation is called kernel. Kernel are grouped into work group - it is set of kernels executed on same streaming multiprocessor. When computation is set on gpu, size of workgroup has to be set and size of final number of threads have to be set.

Memory model

GPU executes many threads simultaneously, so it is not possible to allow all operate with memory. In GPU, three memory space exist. The first is a large global memory. This memory is the biggest memory on GPU (in size of gigabytes) and has the slowest access time of all memory on GPU.

Other memory is the local memory available only to threads in one workgroup; this memory is called local memory and size of this memory is in the order of tens of kilobytes. The access time of local memory is much faster than to the global memory because local memory is on the same chip as the streaming multiprocessors while the global memory is on another chip (because of it's bigger size).

The third memory space is private memory space and in this memory is exclusive only to one thread. This memory is also mapped on registers and is used for variables, counters, etc. This memory has fastest access time but smallest size.

GPU architecture is very different compared to the CPU. The program execution must satisfy some requirements for fast execution. Memory operation has to be coherent or has to be in block. This means that all threads in one workgroup has to read from one memory position or from block of memory. Access time of memory operation is much bigger than on CPU. From this requirements it is clear that for programs with lot of memory operation gpu is not beneficial as for program with less memory operations.

5 Simple GPGPU decomposition

Progressive photon mapping could be divided into several blocks: raytracing, photon tracing, hitpoint illumination and image synthesis.

Raytracing

Raytracing could be understood as sequential examining each pixel's color in final image. Examination routine of one ray is same for each pixel, so it is possible to implement to one kernel examination of one ray. Global work size of raytracing task is equal to number of pixels in final image rounded to size of work group.

Raytracing routine is often written in recursive manner on CPU. This recursive approach is not possible to use on current GPGPU, because GPGPU programming languages do not allow recursive function calling. Therefore, raytracing has to be written without recursion, stack or dynamic array. One possible way of doing this is make iterative function calling (with maximum iteration) and called function will return another ray.

Most scenes are described by set of triangles. There are exists lot of ray-triangle intersection algorithms, chosen technique in own implementation of simple GPU raytracing is Havel's algorithm [3]. When ray is examining with scene, it has to be tested through all triangles in scene.

Solving this problems and combining them into one kernel it is possible to get naive GPGPU implementation of raytracing.

Photon tracing

Photon tracing block uses very similar routines for scene traversal like raytracing. Each initial photon is traversed in separate kernel. The problem occurs when photon has to determine random direction, in photon generating or in photon reflection on diffuse surface. GPGPU does not have any sort of random generator and therefore random generator is needed. It is possible to use classic congruential generator. Same as in raytracing, photon tracing has set max recursion value. This is done, because before starting photon tracing kernel memory to fixed size have to be allocated.

Hitpoint illumination

After each photon tracing pass, new illumination for each hitpoint has to be calculated. It is only computation few formulas if we get all information needed. The slowest thing in whole hitpoint illumination is finding for each hitpoint x photons in radius $R(x)$.

Very naive solution is going through all photons, compute distance from each photons and process only those photons with radius lower than $R(x)$.

Image synthesize

Image synthesize is possible to perform after each photon tracing and hitpoint illumination pass or at the end of all iterations. All needed data are saved along with each hitpoint and therefore computation of luminance is very easy. For each hitpoint will be executed one kernel and this kernel will compute color for his hitpoint. This color will scale by hitpoint weight and atomically add to framebuffer in proper position given by hitpoint framebuffer position.

6 Evaluation of simple GPGPU implementation

Naive implementation was done for evaluating bottlenecks of progressive photon mapping on GPGPU. This naive implementation was done on very simple scene only for experimental usage and for evaluating biggest bottleneck in whole process. Image 2 show results of this simple implementation. Implementation of progressive photon mapping was done on GPGPU and on CPU to evaluate performance between this two execution possibilities.

CPU implementation was evaluated on laptop with Intel i7-4702MQ processor written in c++ and was compile with Intel c++ 15.0 compiler. GPGPU implementation was written in OpenCL and was evaluated on nVidia GeForce GT 750M. Speed of CPU implementation was evaluated using std::chrono library and GPU implementation was evaluated using nvidia nsight timeline profiler.

	GPU	CPU
Raytracing	0.706 ms	1172 ms
Photon tracing	7.897 ms	317 ms
Hitpoint Illumination	2041 ms	1593 ms
Synthesize	0.247 ms	3 ms

Table 1: Performance evaluation between CPU and GPU naive implementation with 320*280 resolution and 100 thousands photons in photon tracing pass

Table 1 and table 2 show performance between GPU and CPU implementation for 320*280 and 1920*1080 resolution. In each photon tracing iteration 100 thousands photons was traced. As it can be seen, GPU is far more

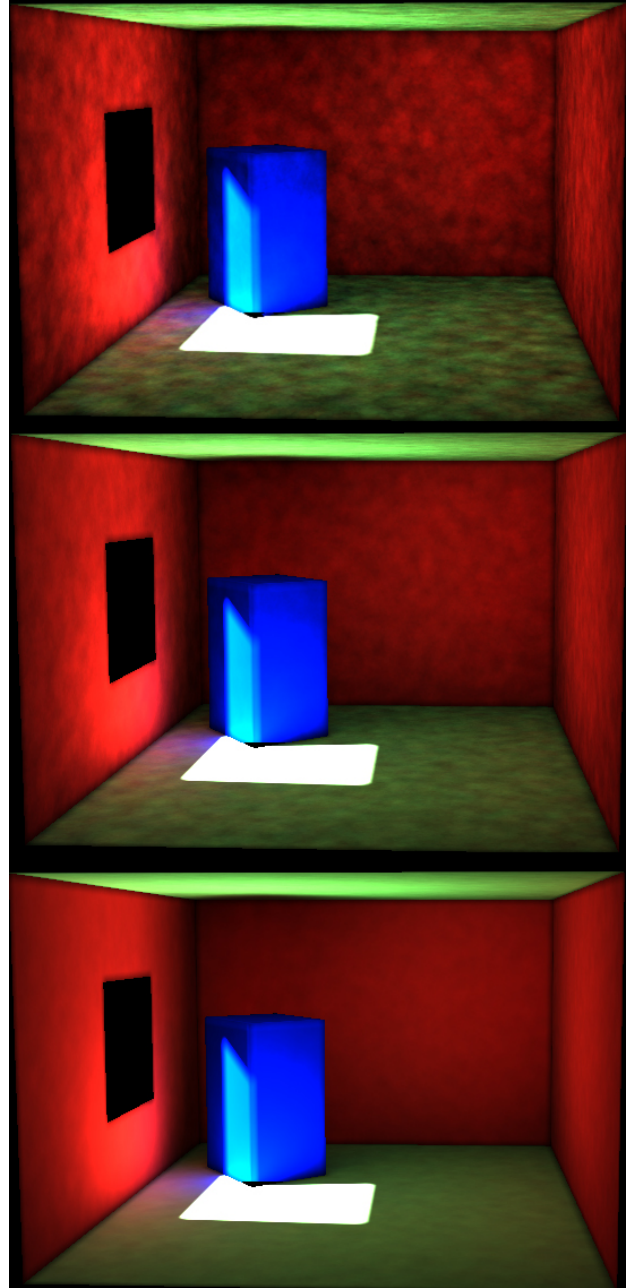


Figure 2: Progressive results of photon mapping. On top image is result with one iteration, on the middle image is result with 10 iterations, on the bottom image is result with 100 iterations. In each iteration 100 thousands photons was generated

faster than CPU implementation in all blocks except hitpoint illumination. CPU block of hitpoint illumination is using kd-tree and if this block will use bruteforce search similar to GPU it will be much more slower. For resolution 320*280 bruteforce on CPU was performed in 68 seconds.

Speed of raytracing and photon tracing is influenced by number of pixels / photons and by complexity of scene. Even when testing scene was simple, photon tracing and

	GPU	CPU
Raytracing	12.107 ms	6 197 ms
Photon tracing	7.897 ms	317 ms
Hitpoint Illumination	38 603 ms	23 957 ms
Synthesize	4.924 ms	91 ms

Table 2: Performance evaluation between CPU and GPU naive implementation with 1920*1080 resolution and 100 thousands photons in photon tracing pass

raytracing aren't bottlenecks because lot of accelerating algorithm for spatial subdivision exists and it is possible to use existing solution for this task for example nvidia OptiX framework. Because of this, this paper will not focus to implementing fast raytracing.

Hitpoint illumination is slowest block in progressive photon mapping implementation on GPU and on CPU. This fact is influenced by very naive implementation of hitpoint illumination and this paper will furthermore focus on accelerating hitpoint illumination on GPGPU.

7 Accelerated photon gathering

Bruteforce implementation of hitpoint illumination is very naive and very slow. In this simple implementation, each thread in workgroup load one photon from memory, compute distance, check if distance is lower than proper radius and optionally accumulate photon flux.

Memory loads are very costly on GPU, even if streaming multiprocessor is multiplexing work between few workgroups, it will wait long time in sleep mode to load data from global memory. One way how to mitigate memory waiting is to use local memory.

This section will present accelerating approaches to accelerate this task. Mostly every other approach will be based on previous approach.

Local memory

Idea is to load block of photons into local memory and go through this photons from local memory. Access to local memory is faster than access to global memory, so it should lead to acceleration.

Each thread in workgroup will load one photon into local memory. Kernel driver will recognize this as block loading and this block loading should be done faster than sequential load of each photon.

Photon sorting

When illumination is computed for hitpoint x , in computation have to be included only those photons lying on same mesh as hitpoint x . When illumination is computed by bruteforce, in hitpoint illumination photons from all mesh are included and when photon from other mesh is loaded

to illumination computation, this photon are discarded and costly load was in vain.

One way, how to solve problem with loading photons from different meshes, is to sort photons into unique photon map for each mesh. This task should be done in separate kernel. Each thread will load one photon, check its mesh, by special operation called atomic inc will get position in separated photon map and this photon will save into proper position in memory.

When hitpoint is illuminated it will check hitpoint mesh look into proper photon map and will sequentially loading photons from proper photon map. In this case it is not possible to use local memory because it is not guaranteed that all hitpoints in workgroup will lie on same mesh.

Hitpoint sorting

It is needed to ensure that all hitpoints in workgroup lie on same mesh to use coherent approach (all thread are reading from same memory location) to fast loading from memory.

Hitpoint sorting could be done on CPU side because this work will done only once per whole render process. Hitpoints are loaded to CPU side after raytracing pass. This hitpoints are sorted to separate arrays. In the end of each hitpoint array, special filler hitpoints have to be saved.

This is because ending workgroup of one hitpoint array is not filled fully, for instance we have size of workgroup 256 and in last workgroup of hitpoint array is only 150 hitpoints, so if we do not save 106 filler hitpoints, this last workgroup will consist from hitpoints from two meshes.

If it is ensured that hitpoints in all workgroup is same, performance should accelerate because all threads in workgroup is loading sequentially from same memory.

Hitpoint sorting and local memory

If all photons are separated and similar hitpoints are saved in one workgroup it is possible to use local memory to save photons from separated photon map. This should lead to best acceleration and should be much faster than naive sequential loading.

Hitpoint clustering and spatial grid

Last proposed acceleration is to build spatial grid on each scene objects and sort photons to appropriate subspaces. Then all hitpoints will search only through limited space and will not go through all photons on one mesh. For speedup purposes photons in each workgroup photons should be as close as possible, so some sort of clustering is needed.

8 Experimental results

Acceleration approaches introduced in previous section was implemented on same scene. Table 3 shows result

of this implementation.

	320*280	1920*1080
Naive solution	2041 ms	38 603 ms
Local memory	1109 ms	17 750 ms
Photon sorting	611 ms	10 539 ms
Hitpoint sorting	484 ms	8 384 ms
Hitpoint sorting, local memory	327 ms	5 422 ms

Table 3: Performance evaluation between all proposed acceleration techniques.

As it can be seen, all proposed approaches lead to some sort of speedup. Local memory accelerate computation nearly twice. Photon sorting accelerate naive solution nearly four times. Using this approach special kernel execution is needed. Kernel for photon sorting has been executed in nearly 1ms, so final speedup of hitpoint illumination is much higher than overhead made by special kernel execution.

Hitpoint sorting uses photon sorting with sorted hitpoints on CPU side, so some CPU overhead is needed. Because this task is performed only once - after raytracing pass - this overhead does not matter. Last accelerating approach use local memory with hitpoint sorting and photon sorting. This approach is much faster then all previous approaches.

All of this test was written in OpenCL. OpenCL has built-in function distance() and even when this function should be very fast, it isn't. When on last, fastest approach was distance() function replaced by manual distance calculation from high school, this block get nearly three times speedup! Hitpoint illumination with hitpoint sorting and local memory extended by manual distance calculation has 125 ms execution on 320*280 resolution and 1 941 ms execution on 1920*1080 resolution.

Hitpoint clustering and spatial grid

For clustering, k-means algorithm was used. K-means algorithm return hitpoints clustered in close clusters. One problem occurs with this type of clusters, it is not possible to made fix number of hitpoints in one cluster so lot of filler hitpoints have to be used to fill gaps in hitpoints so one workgroup will only consist from hitpoints from one cluster and eventually filler hitpoints. Lot of filler hitpoints have to be used with k-means. For 1920*1080 resolution nearly 33% filler hitpoints (in sum of all hitpoints) was generated and this lead to slower performance without using grid.

Image 3 shows how clusters are made in scene. For each object in scene, simple (naive) grid structure is proposed. First bounding box of object is computed, then longest axis is split by fixate number of pieces. Length of one divided piece give length of cube of one subspace and then this subspaces are uniformly distributed on object. For each subspace on each object maximum size (in

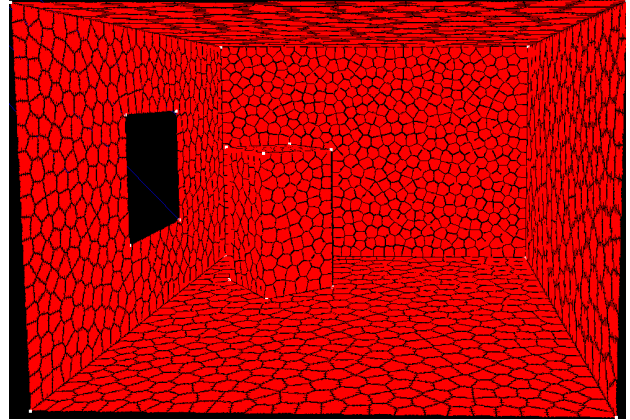


Figure 3: Hitpoint clusters made by k-means

photons count) have to be set. This approach lead to easy implementation and this approach has nice modification options.

	320*280	1920*1080
Hitpoint sorting, local memory	172 ms	2 852 ms
Grid structure	87 ms	730 ms

Table 4: Performance evaluation of naive grid structure with clustered hitpoints.

Table 4 shows that by using clustered hitpoints and naive spatial grid it is possible to achieve double acceleration.

9 Conclusion

This paper described an experimental implementation of Progressive photon mapping by its decomposition into several blocks suitable for GPGPU. More or less naive implementation of these blocks in GPGPU was proposed as well. The full Progressive photon mapping performance in GPGPU was tested and compared to the CPU. The slowest block turned out to be Hitpoint illumination and for this block, an acceleration approaches were proposed as well.

The Fastest evaluated approach was sorting photons into a mesh, sorting hitpoints into a mesh as well, compute clusters on each mesh and use grid for photon acceleration search. Used grid is very naive and in future work this grid should be replaced by more complex and efficient grid, but this very naive grid with clustered hitpoints shows way to accelerate hitpoint illumination block. Another thing in future work should aim to efficient clustering with low filler hitpoint ratio.

The future work also includes the overall profiling of the GPGPU Progressive photon mapping implementation and also various planned improvements in quality of rendering and speed.

References

- [1] Toshiya Hachisuka and Henrik Wann Jensen. Stochastic progressive photon mapping. *ACM Trans. Graph.*, 28(5):141:1–141:8, December 2009.
- [2] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *ACM Trans. Graph.*, 27(5):130:1–130:8, December 2008.
- [3] Jiri Havel and Adam Herout. Yet faster ray-triangle intersection (using sse4). *IEEE Transactions on Visualization and Computer Graphics*, 2010(3):434–438, 2010.
- [4] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag.
- [5] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.
- [6] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. 1993.
- [7] Eric Veach and Leonidas Guibas. Bidirectional Estimators for Light Transport. 1994.
- [8] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

Efficient Implementation of Bi-directional Path Tracer on GPU

Bc. Vilém Otte*

Supervised by: RNDr. Marek Vinkler Ph.D.[†]

Faculty of Informatics
Masaryk University
Brno / Czech Republic

Abstract

Most of the implementations solving photo-realistic image rendering use standard unidirectional path tracing, having fast and accurate results for scenes without caustics or hard cases. These hard cases are usually solved by a bidirectional path tracing algorithm. However, due to the complexity of the bidirectional path tracing algorithms, its implementations almost exclusively target sequential CPUs. The following paper proposes a new parallel implementation of the bi-directional path tracing algorithm for the GPUs. Our approach is compared with existing solutions in terms of both performance and image quality. As the references we use the standard unidirectional GPU path tracer and commercial off-line bidirectional path tracers. We achieve interactive rendering rates for scenes of medium complexity.

Keywords: bidirectional path tracing, path tracing, physically based rendering, ray tracing, CUDA

1 Introduction

Global illumination research recently focuses on unbiased methods. Unbiased method is such method, that under some assumptions, converges to the solution of the rendering equation.

One of the common techniques for rendering unbiased images is path tracing, respectively the extension of path tracing - bidirectional path tracing. These techniques are well known for high quality rendering output. Formerly, most of the path tracing and bidirectional path tracing implementations were done on the CPU. However, these techniques are well fit for massively parallel hardware, like the recent generations of graphics hardware.

Even though there are already present implementations of bidirectional path tracing on the graphics hardware, most of the implementations are either using the graphics hardware only for small part of the computation or they do not contain any surface shading at all.

The presented work focuses mainly on the implementation of a full featured bidirectional path tracing renderer,

allowing for advanced materials, including textures and computing the entire image using the graphics hardware. The contributions of this paper are the following:

- Design and implementation of three variants of bidirectional pathtracing on the GPU.
- Comparison of their speed and quality to the ground truth path tracing solution.
- Comparison to other GPU based renderers.

2 Related Work

In this section we summarize literature most relevant to bidirectional path tracing - various approaches to global illumination focused on parallel GPU algorithms.

2.1 Ray Casting

The core of almost every global illumination technique is ray casting, which allows for finding closest ray-primitive intersection point along each ray and also to test visibility between two points. This visibility computation is the key to the evaluation of light transport. The focus of research on ray casting are following three issues: selection of acceleration structure, their construction and their traversal.

High performance and well established ray casting solutions are available for public, including NVidia OptiX [11] targeting NVidia hardware (which is actually full ray tracing engine), Intel Embree [15] targeting traditional SIMD-based CPU architectures. Our implementation is based on the open source framework by Aila et al. [1, 2].

2.2 Early Global Illumination

Whitted [14] used ray-casting for generation of photo realistic images, allowing for recursive specular reflections and refractions. Later Cook extended ray tracing to distributed ray tracing to allow for effects such as diffuse interreflection [3]. The rendering equation was introduced in 1986 by Immel et al. [5] and Kajiya [7].

The rendering equation describes light transport in the scene. Light transport describes the energy transfers in a given scene that affects visibility.

*vilem.otte@post.cz

[†]xvinkl@fi.muni.cz

2.3 Path Tracing and Bidirectional Path Tracing

Path tracing algorithm allows for computing the solution of the rendering equation. By computing the paths of light from the camera into the scene (eventually reaching light source), it closely resembles the behavior of light. The path starts with primary ray at the camera and is traced into the scene. Upon intersection, it continues in random direction.

For further efficiency, at each vertex of the path it is determined whether path should be terminated or not by Russian roulette, thus preventing infinite path lengths. It allows for physically based computation of lighting, and so the synthesis of photo realistic images. The algorithm is also unbiased, with theoretically absolute accuracy when using infinite number of samples.

Bidirectional path tracing is further extension of path tracing algorithm, introduced by Veach [12]. By generating one sub path from a light source and one sub path from the camera, later joining them together, it is possible to handle indirect lighting computation more robustly (and efficiently) compared to ordinary path tracing. Furthermore, this modification still keeps the resulting algorithm physically based and unbiased.

2.4 Other Global Illumination Methods

Lately a lot of fast algorithms for computation of global illumination were introduced. Most of the algorithms were, compared to path tracing, biased, providing a trade off between rendering quality and speed.

Recent interactive global illumination methods are modifications of Virtual Point Lights, introduced by Keller et al. [9], allowing for real time smooth global illumination [10]

One of the other popular global illumination techniques, used in real time rendering, are Cascaded Light Propagation Volumes [8].

Photon mapping introduced by Jensen [6], is currently popular in production renderers, especially for interior rendering. The technique was further extended into progressive photon mapping [4]. Compared to standard photon mapping, which is a biased rendering algorithm, progressive photon mapping is an unbiased one.

3 Bidirectional Path Tracing

This section presents some improvements when using bidirectional path tracing. For proper description of these features it is critical to define some terms.

Naive Path Tracing This designation is used for path tracers that does not do any explicit steps, but wait for camera ray to actually hit light (or get terminated).

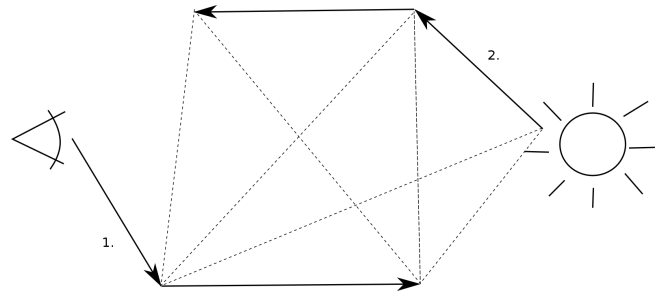


Figure 1: All possible connections between a light path and camera path. 1. Represents camera path, 2. Represents light path, Dashed lines represent possible connections between light path and camera path.

Standard (Explicit) Path Tracing Due to very bad convergence ratio using naive path tracing, it is often understood that generic path tracing algorithm does single explicit step towards light on each vertex of the path. These path tracers are to be designated as standard, or explicit, path tracers.

For the sake of completeness, pseudo algorithms for standard path tracing (see Algorithm 1) and bidirectional path tracing (see Algorithm 2) are provided.

Bidirectional path tracing, computes two paths, one from the light and another one from the camera. These paths are later connected (see Figure 1). These connections attempt to solve several problems introduced by unidirectional path tracers:

Small light sources For naive path tracers, the probability of hitting a light source is proportional to its size. Having point lights (lights that are infinitely small) in naive path tracer often ends up with nothing visible in rendered image, as the probability of hitting the light source reaches zero.

This can be solved by sampling light in explicit manner each step in path computation, resulting in very fast convergence for directly lit scenes. While sometimes this technique is referred as explicit path tracing, it actually is a special case of bidirectional path tracing, where the light path contains only a single vertex.

The visibility function between each camera path vertex and light path vertex has to be computed, resulting in $2 \cdot N$ cast rays for camera path length of N .

For bidirectional path tracing, even more complex paths from the light can be easily evaluated, e.g. light hidden inside a lamp.

Interior scenes lit by exterior light Assuming we are inside a room, where there is only a single window, standard, or even explicit, path tracers are not going to converge very quickly because the probability of sampling the light is low. In fact, in case where no path vertex on camera path lies in direct light, the contribution of that path is

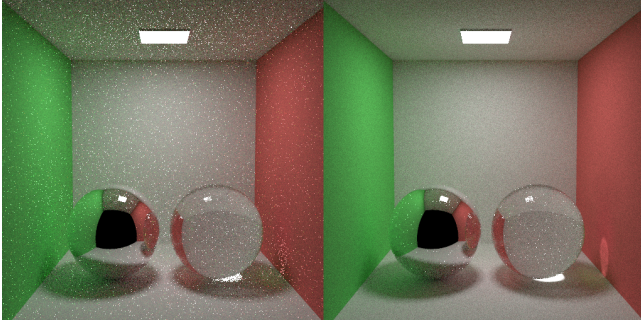


Figure 2: Difference between path tracer and bidirectional path tracer when computing caustics. On the left side, the fireflies generated by standard path tracer compared to more smooth caustics generated by the bidirectional path tracer on the right side.

zero. This is one of the so called 'pathological scenarios' for path tracing, where the algorithm fails to provide result fast enough.

Using a bidirectional path tracer with light path of length M , it is enough that single point of the light path is directly visible from any of the camera path vertices. If this condition is true, that paths' contribution will be non-zero, resulting in faster convergence.

Caustics convergence Using a unidirectional path tracer often results in poor caustics and fireflies in the resulting image, as their convergence rate is by magnitudes worse to convergence rate of diffuse light. Computing caustics with bidirectional path tracer is faster (see Figure 2), yet there are still difficulties with reflected caustics. Bidirectional path tracer can be, however, further extended with Metropolis Light Transport [13], improving the performance on reflected caustics.

Algorithm 1 Path Tracing

```

1: procedure PATHTRACE
2: for each path:
3:    $ray \leftarrow$  setup primary ray
4:   while  $ray.terminated = false$  do
5:      $result \leftarrow raycast(ray)$ 
6:     if  $result.hit = false$  then
7:       Accumulate background color
8:        $ray.terminated \leftarrow true$ 
9:     else
10:      Compute and Accumulate surface emission
11:      Compute contribution of random light
12:      if Contribution is non zero then
13:        Accumulate contribution
14:      if Russian roulette terminates path then
15:         $ray.terminated \leftarrow true$ 
16:      else
17:         $ray \leftarrow$  Get  $B*DF$  Sample

```

Algorithm 2 Bidirectional Path Tracing

```

1: procedure BIDIRPATHTRACE
2: for each path:
3:   Generate vertex on random light
4:   Push this vertex to light path
5:    $ray \leftarrow$  setup light ray
6:   while  $ray.terminated = false$  do
7:      $result \leftarrow raycast(ray)$ 
8:     if  $result.hit = false$  then
9:        $ray.terminated \leftarrow true$ 
10:    else
11:      Push this hitpoint to light path
12:      if Russian roulette terminates path then
13:         $ray.terminated \leftarrow true$ 
14:     $ray \leftarrow$  setup primary ray
15:    while  $ray.terminated = false$  do
16:       $result \leftarrow raycast(ray)$ 
17:      if  $result.hit = false$  then
18:        Accumulate background color
19:         $ray.terminated \leftarrow true$ 
20:      else
21:        Compute contribution by joining light path
22:        with this vertex
23:        if Russian roulette terminates path then
24:           $ray.terminated \leftarrow true$ 
25:        else
26:           $ray \leftarrow$  Get  $B*DF$  Sample

```

3.1 Parallel Bidirectional Path Tracing using GPU

Computation of different samples, (sub)paths in terms of path tracing, and different pixels is independent on each other. Path tracing and also bidirectional path tracing are therefore good candidates for massively parallel computation.

The resulting parallel algorithm looks similar to sequential version. The parallel run is performed over the pixels, with each sample is computed by a single thread in a single kernel launch.

4 Implementation

The bidirectional ray tracing kernels were implemented on top of open source framework by Aila. For this purpose, new rendering kernels for each variant of bidirectional path tracer were added.

To achieve high performance of the source code, the speculative while-while traversals were used. As opposing to persistent threads they perform, in general, better on new hardware. Also, bounding volume hierarchy with spatial splits was used as acceleration structure for the rendering.

The Aila framework was further extended to support high resolution textures, reflective, refractive and dielec-

tric materials. Light sources are handled as geometry with emissive material, so in general any number of light sources is supported.

The bidirectional path tracer kernel always processes one pixel in single walk through, e.g. the light path generation (storing required data), followed by computation of the camera path. During each step of the camera path computation the join for currently processed vertex is performed when necessary.

5 Optimizations and Limitations

The following section describes possible optimizations and limitations when implementing bidirectional path tracer on the GPU. For each optimization a brief summary is given.

5.1 Sub path join

Full join

Joining of the light sub path of length M and the camera sub path of length N is a non trivial task. There are multiple ways to join these, the intuitive solution is joining each vertex from light sub path to each vertex in camera sub path. While this actually leads to faster convergence, $N \cdot M$ rays have to be used to compute the visibility between samples, which is slow. Moreover, the full light path has to be stored in the memory, and so larger memory space is required.

Single-step join

Performance wise, the most efficient idea is joining the last camera sub path vertex and the last light sub path vertex. Such approach has some advantages. Single path computation is of the same performance as naive path tracing, although improving the situations where naive path tracing has major problems. However, by combining only the ends of both paths contributions of these light paths are very small.

K-step join

It is also possible to select an approach using randomized algorithm. For each of the $N \cdot M$ joining rays, the algorithm discards some of these joins. The actual joining ray rejection can be built upon multiple criteria - either fully random, or deterministic (removing less contributing joins and accordingly weighting the rest). This join is to be designated as K-step join. The join is performed by taking each vertex from camera against k vertices from the light path. The k value has to be smaller than the number of vertices in the light path.

When implemented properly, the different sub path joins do not break the unbiased property of the algorithm.

5.2 Path pre-generation

Given a static scene, all the light paths can actually be pre-computed. Later, during the execution of the algorithm, we only select one of the light paths from given M pre-computed light paths.

While this approach is highly efficient, it often means that the resulting algorithm is biased. This can be overcome by re-generating these light paths on runtime. Once we start joining samples to random pre-generated paths, it is possible that some samples are to be joined with a single light path. This could lead to unnatural patterns in resulting image, and of course breaking unbiased nature of the algorithm. By re-generating the paths after they have been used, it is possible to avoid this problem.

Unless large M is selected, the quality of resulting image can be highly degraded. To keep the quality of the resulting image high enough, it was experimentally evaluated that the number of light paths must be at least the same as the number of pixels in the resulting image.

5.3 Biasing

Biasing the algorithm does not have much sense for simulations. Although, for performance heavy applications, in case where we have limited time for calculating an image, for example in games, it is possible to trade off quality for speed.

During the implementation, two of the biasing techniques were considered.

Limiting maximum camera path length Generally, the camera path length can be very long (assuming it doesn't directly hit the light), until Russian Roulette finally terminates the path.

By limiting the length of camera path to some value it is possible to increase performance. First of all, longer paths generally tend to have smaller contribution, by ending them at some given maximum length we terminate them early, effectively reducing the number of computations they need to do.

Also, from the GPU perspective, we are always waiting for the longest path to finish the computation, in the worst situation, whole warp is waiting for single thread to finish a very long path. By bounding the maximum length, we effectively reduce this issue and increase the computation performance.

On the other hand, the unbiased nature of the algorithm is lost, by limiting the maximum path length we are effectively limiting the maximum number of reflections/refractions, which may cause visible problems in images (for example, missing reflection).

Random Number Generation As each Monte-Carlo technique, path tracing and bidirectional path tracing, heavily depend on random number generation. Having a

random number generator with large period means, that resulting image will converge closely to the ground truth.

Each Monte-Carlo technique spends some amount of time in random number generator. If the application targets performance instead of precision, it is possible to pre-generate random numbers into an array and use those later.

Doing so introduces a limit at which the image is not able to converge more towards the ground truth (as all the samples were already taken).

It is important to note that the results taken using the implementation were recorded with unbiased version of the algorithms. The biased version is between 2 and 3 times faster compared to unbiased version.

6 Results

6.1 Evaluation and Analysis

Results of the implementation were evaluated on low-end laptop GPU NVidia GeForce 720M, with 1.5 GB memory. The system was running under Windows 8.1 OS, with CUDA 5.5 installed. Kernels were compiled with compute capability 2.0. The light paths were generated on the runtime.

A low end GPU was used to demonstrate, that even current generation laptop based GPUs are capable of interactive rendering of moderately complex scenes.

The rendered images were taken each 5 seconds, while the used algorithm was running progressively. Results presented in this section show the difference in quality between the specific implementations. The resulting images are also compared to ground truth using the root mean square error (further RMSE, lower is better).

The first set of comparisons is between path tracing and bidirectional path tracing. Followed by comparison against other GPU based Monte-Carlo rendering systems.

The single-step join does need to keep just a single (last) vertex of the light path. K-step join needs to keep K vertices of the light path in the memory during the computation, while full join needs to keep all the vertices in the light path. While single-step join and K-step join have constant memory footprint, the full join footprint grows with the length of the light path.

6.2 Bidirectional vs. Unidirectional

Cornell Box

We ran two algorithms on the following scene, bidirectional path tracing (with full path join) and standard path tracing. Both of the resulting images are compared to the ground truth. The sample images were taken after 5 seconds and after 10 seconds. (see Table 1)

From the given comparison, it can be stated that bidirectional path tracing with full path joining converges faster

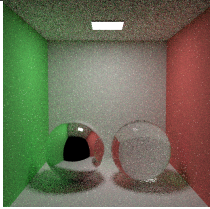
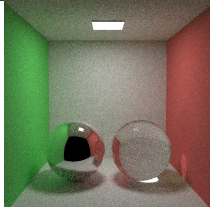
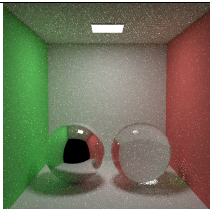
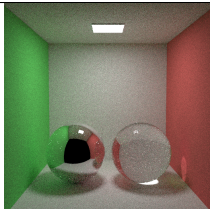
	Path Tracing	Bidirectional
5s		
RMSE	0.1002	0.0628
10s		
RMSE	0.0784	0.0456

Table 1: comparison between Path Tracing and Bidirectional Path Tracing with full path join.

compared to standard path tracing. This is especially the case for caustics.

Crytek Sponza

The following scene shows complex materials, like textured, alpha-tested, reflective and refractive surfaces, lit by an area light source.

All three different bidirectional kernels were run on this scene, and the results show how the scene looked after 5 seconds of processing. All the results are compared to the ground truth in terms of RMSE (see Table 2).

We observed the same behavior in all of our measurements, full path join results in best image quality and fastest speed. This shows, that the algorithm is not bound by memory performance, in which case K-step join or single step join would result in higher performance and thus quality on a fixed budget.

Caustics Test

We also ran three different bidirectional kernels, as well as standard path tracing method, on a scene with a caustic. The results show how the scene looked after 5 and 10 seconds of processing (see Table 3).

The full path join also results in the highest quality caustics, that are almost perfectly smooth after ten seconds of computation.

6.3 Other GPU rendering packages

iRay

NVidia iRay is a physically based renderer highly scalable in performance across GPUs and CPUs. We rendered the Crytek Sponza scene using our bidirectional path tracing renderer and the iRay from a similar viewpoint (see Table 4).

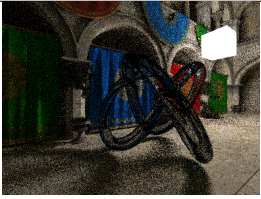


Renderer	Image	RMSE
Single-step		0.1584
K-step (K=3)		0.0701
Full		0.0095

Table 2: Equal-time comparison between three different implementations of bidirectional path tracing and their distance to ground truth in terms of RMSE.

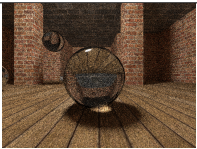
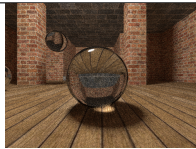
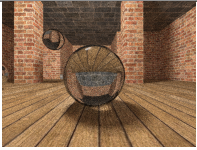
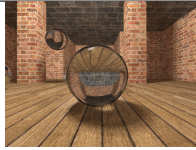

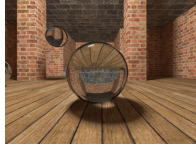
Renderer	5s	10s
Single-step		
K-step (K=3)		
Full		

Table 3: Comparison of a caustic scene with three different implementations of bidirectional path tracing. The contrast was intentionally enhanced so it is possible to see the difference in sampling inside caustics casted by a glass sphere. Single step join has very slow convergence and so the resulting image is darker compared to the others.


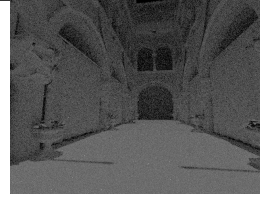


	NVidia iRay	Bidirectional
5s		
10s		

Table 4: Comparison between iRay and our bidirectional path tracing implementation. The brightness/contrast difference is caused by different handling of output between both implementations.

Again we target images generated after 5 and 10 seconds, which shows the quality achievable using an interactive preview on low end graphics card.

The resulting images are untextured, as there is not a full support for textured surfaces in iRay, and the scene was lit using single directional light. This setting also allows for an easier comparison of the quality of the global illumination without the masking effect of the textures.

The technique iRay uses is actually standard path tracing, in comparison it is clearly visible that our Bidirectional technique produces smoother results for a similar time budget.

LuxRender

LuxRender is a physically based and unbiased rendering engine. The LuxRender package was used through Blender software, possibly limiting some options leading to slightly decreased quality (see Table 5).

LuxRender uses a technique called LuxRays to produce the image, it is an unbiased technique similar to bidirectional path tracing. The implemented bidirectional solution might seem to converge better, although it is important to state, that LuxRender is a very large package supporting complex material setup (along with subsurface scattering effects for example), while the implemented solution does not.





	LuxRender	Bidirectional
10s		
20s		

Table 5: Comparison between LuxRender and our bidirectional path tracing implementation. The brightness/contrast difference is caused by different handling of output between both implementations.

7 Summary

Path tracing, in general, excels in exterior scenes or when there are large light sources. Bidirectional path tracing is an extension that allows for faster computation of more complex situations, like interior scenes or small light sources.

The created GPU-based implementation showed, that bidirectional path tracing is also suitable for massively parallel implementation. The results confirmed, that convergence rates are in general better for bidirectional path tracer, which effectively reduces computation time.

Full path joining proved to lead to the highest quality global illumination, although keeping large memory footprint. However, on modern GPU architectures, the memory footprint required by full path joining is bearable.

K-path join proved to be an interesting alternative to full path join. Even though the full path join actually results in better convergence rates, it is possible to alter the number of joins for K-join on the fly. This can be used to achieve a constant refresh rate when running the algorithm in progressive mode. Such approach can be interesting for interactive preview of rendered scene.

References

- [1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM.
- [2] Timo Aila, Samuli Laine, and Tero Karras. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. NVIDIA Technical

Report NVR-2012-02, NVIDIA Corporation, June 2012.

- [3] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 137–145, New York, NY, USA, 1984. ACM.
- [4] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *ACM Trans. Graph.*, 27(5):130:1–130:8, December 2008.
- [5] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. *SIGGRAPH Comput. Graph.*, 20(4):133–142, August 1986.
- [6] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag.
- [7] James T. Kajiya. The rendering equation. In *Computer Graphics*, pages 143–150, 1986.
- [8] Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 99–107, New York, NY, USA, 2010. ACM.
- [9] Alexander Keller. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.
- [10] Jaroslav Křivánek, Iliyan Georgiev, Toshiya Hachisuka, Petr Vévoda, Martin Šik, Derek Nowrouzezahrai, and Wojciech Jarosz. Unifying points, beams, and paths in volumetric light transport simulation. *ACM Trans. Graph.*, 33(4):103:1–103:13, July 2014.
- [11] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, July 2010.
- [12] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford, CA, USA, 1998. AAI9837162.
- [13] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 65–76, New York,

NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

- [14] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.
- [15] Sven Woop, Louis Feng, Ingo Wald, and Carsten Benthin. Embree ray tracing kernels for cpus and the xeon phi architecture. In *ACM SIGGRAPH 2013 Talks*, SIGGRAPH '13, pages 44:1–44:1, New York, NY, USA, 2013. ACM.

Rendering Large Point Clouds in Web Browsers

Markus Schütz*

Supervised by: Michael Wimmer†

Institute of Computer Graphics
Vienna University of Technology
Vienna / Austria

Abstract

We present a method to display large point data sets in web browsers. Such data sets can consist of hundreds of millions to billions of points and are therefore too large to be loaded and rendered at once. The idea of this method is that only points inside the view frustum and up to a certain level of detail are loaded and rendered. Using adjustable point-count limits, even mobile and low-end desktop hardware is able to display these point clouds in real time or at least interactively. The implementation is based on standard web technologies and requires no additional plugins to be installed. This allows developers to combine it with other web applications, like mapping software, in order to synchronize georeferenced point clouds with world map overlays. We also show how to choose point sizes in order to avoid holes and hide varying point densities caused by different levels of detail.

Keywords: WebGL, Point Cloud, LIDAR

1 Introduction

Various 3D scanning technologies such as laser scanners and photogrammetry produce enormous amounts of point cloud data. Datasets with billions of points are not uncommon anymore. Processing and rendering these datasets is a challenging task that neither the memory nor the speed of today's hardware can handle in real time unless broken down into smaller parts. The simplest approach is to tile datasets in small regional chunks and handle one or a few at a time. Another one is to subsample data down to a manageable size. A third one, which is gaining more and more popularity, is the combination of both by creating a multi-resolution hierarchy. Such a hierarchy consists of multiple levels of some sort of tree with a low-resolution model stored at the top and increasing resolutions stored in all descendants.

This paper is based on the multi resolution octree of Instant Points [1]. The concept of inner and outer octree was simplified to one octree where all nodes, including root, inner and leaf nodes, store various resolution subsets

of the original point cloud data. Unlike the inner octree approach, we do a uniform point selection with a user-defined point spacing. This approach has higher preprocessing costs but improves visual quality at lower levels of detail. This is especially useful with slow internet connections, where viewers have to wait longer times until higher levels of detail are loaded, but also for low-end hardware with a low point budget. The predefined point spacing also allows for a new adaptive point-size mode that changes the point size according to the level of detail in order to avoid holes. However, points that are close together will be discarded due to the spacing requirements. Like the Instant Points system, we store one file for every node, which makes it possible for the client to load the necessary data without relying on server-side applications.

Our implementation, Potree [2], and some of the examples presented in this paper are available online at <http://potree.org>

2 Related Work

QSplat [3] is a multi-resolution algorithm that traverses a bounding-sphere hierarchy and builds the rendered point cloud point by point. It adjusts level of detail to the rendering duration. The fine granularity of the hierarchy, each point is represented by a leaf or inner node, allows to render low-detail images during user navigation and progressively higher levels of detail once movement has stopped. Hierarchy traversal, on the other hand, is very costly, and the point-wise assembly of the visible dataset makes it hard to efficiently use the GPU.

Instead of associating each node with just one point, Layered Point Clouds (LPC) [4] store M points per node, where M can be chosen freely. This approach is much more GPU friendly, as points can be stored on the GPU in blocks of M points each, and the application only has to tell the GPU which blocks to render. Our approach also stores blocks of points in each node. The difference to LPC is that points are chosen differently and it is not required to have exactly M points in each inner node. LPC also uses a binary tree that continuously splits along the longest axis, while our approach uses an octree instead.

Plas.io [5] is a web-based viewer for LAS and LAZ files, and coupled with the point-streaming back-end grey-

*mschuetz@potree.org

†wimmer@cg.tuwien.ac.at

hound. The main differences to our implementation are that greyhound uses a quadtree instead of an octree, points are chosen according to the distance to a grid center during indexing, and the client does not depend on the tree hierarchy since the server takes care of it. The quadtree approach works well for LIDAR datasets with large length and width but relatively low height. General-case point clouds with larger height are problematic, though. Indexing, on the other hand, is faster due to simpler point-selection methods.

glob3mobile [6] is a mobile mapping framework with point cloud support. Points can be stored either in a quadtree or in an octree, depending on the extent of the dataset. Data is stored in leaf nodes only, which decreases potential overhead in high zoom levels and top down views, where few leaf nodes are visible. On the other hand, at low zoom levels with a high amount of visible nodes, the overhead increases.

ShareLiDAR [7] also uses a multi-resolution model approach and it is currently the only web viewer listed in this paper that supports normals and therefore illumination as well. According to their description, they have a preprocessing throughput of 40kb of LAS files which is roughly 1500 points per second. For large datasets with billions of points, this throughput is problematic.

3 Multi-Resolution Octree

In order to be able to render point clouds in real time and to keep load times low, we first create a multi-resolution octree hierarchy of the point cloud. The first level of this octree, the root node, contains a coarse representation of the whole point cloud. The resolution is defined by the *spacing* parameter, which specifies the minimum distance between points at root level. The default spacing is set to

$$spacing = \frac{boundingBoxDiagonal}{250} \quad (1)$$

but users may define other values if required. Each subsequent level halves the spacing and stores models with increased resolution.

The spacing influences the number of points in each node and therefore affects download times, the total number of nodes, the number of rendered nodes for a certain point budget and the efficiency of frustum culling. The default value was chosen as a trade-off between fewer points per node to improve download times but enough points to avoid generating a large amount of mostly empty nodes.

Since hierarchy traversal is done on client side, the client has to know about the hierarchy. Depending on the size of the input dataset and the indexing parameters, the octree can grow up to millions of nodes. Downloading the full hierarchy at once increases the initial load times. To keep load times low, the hierarchy is split into smaller chunks, and only parts that are needed will be loaded. A chunk contains a node and its descendants for the next

chunkDepth levels. Assuming *chunkDepth* = 5, a chunk is generated for the root node, containing the hierarchy from root to descendants at level 5. The same is repeated for all nodes at level 5, resulting in a multitude of chunks that contain the hierarchy from level 5 up to level 10 in the respective regions.

4 Point Cloud Indexing

This section describes how a multi-resolution hierarchy is created from an input point cloud.

First, the bounding cube of the input data is calculated and the spacing and the depth of the octree are defined. The default value for spacing is given by Equation 1 but may be set to any other value by the user. The octree depth has to be defined by the user. In the next step, points of the input dataset are subsequently added to the octree. If the distance to any other point inside the root is larger than the spacing, the point is added to the root node. If there is already another point in close proximity, it is passed to the next level and the same test is repeated with half the spacing. This process is repeated until the point has been added to a node or the octree depth has been exceeded. In the latter case, the point is discarded and will not be added to any node. No duplicate points are generated.

Each node may contain thousands of points. To reduce the amount of necessary distance tests while adding a new point, points in a node are stored in a 3D grid. The length, width and height of each cell is equal to the *spacing* at the nodes level. During the distance test, only points in the same cell and neighbouring cells have to be tested.

The hierarchy and nodes are written to disk regularly, e.g., for every millionth point processed. The results up to the last write can be viewed at any time, giving the user the possibility to immediately cancel the conversion process if, for example, the user decides to adjust conversion parameters based on the current results.

Each node of the octree is given an ID that also represents its exact position in the hierarchy. The numbers in the ID stand for indices ranging from 0 to 7. The root node is named *r*. The first child of the root node has index 0 and therefore its ID is *r0*, while the last child of the root has index 7 and ID *r7*. The same is repeated for each level, always concatenating the ID of the parent and the index of the child to calculate the child's ID. The second child of *r0* has index 1 and thus the id *r01*.

5 Data Streaming and Disposal

This section describes how point data is loaded and unloaded.

The indexing process splits the point data into small nodes and stores each of them in its own file. The only task of the server is to host these files and send them to the client upon request.

Initially, the client loads metadata such as bounding box, spacing and point attributes from the server. In the next step, the first hierarchy chunk, containing the hierarchy for the first few levels, is loaded. At this point, the client starts to calculate the visible nodes and determines nodes that are visible but have not yet been loaded. Unloaded nodes with the largest screen-projected size are then requested from the server until at most *maxParallelRequests* are loaded at the same time. A value of 4 for *maxParallelRequests* has proven to work well in practice. If a node has a hierarchy chunk associated with it, that chunk will be loaded as well, thus expanding the currently loaded hierarchy by another few levels in the respective region.

The client cannot load and store an infinite amount of data in memory. It is therefore necessary to remove nodes which are no longer or rarely used. This is done by keeping track of the least recently used (LRU) nodes. After a certain threshold on the number of loaded points has been reached, the client starts to remove least recently used nodes from memory before loading new ones. Previously disposed data is often loaded faster on future http requests since web browsers usually cache data by themselves.

6 Rendering

This section describes the rendering process, including hierarchy traversal to calculate visible nodes, coloring, calculating point sizes and different point-rendering qualities.

6.1 Octree Traversal and Visible Node Determination

Octree traversal fulfils 4 main tasks:

- Discard nodes outside the visible area (Frustum Culling)
- Prioritize nodes with large screen-projected size
- Enforce point budget
- Discard nodes with a small screen-projected size

The traversal is done in a largest to smallest screen projected size order since nodes with a larger projected size tend to have a higher impact on visual quality. This is done by adding the children of each traversed node into a priority queue, and then visiting the node with the highest priority. The priority is given by the screen-projected size. A child always has a lower priority than its parent, but distant high-level nodes may have a lower priority than low level nodes that are close to the viewer. During traversal, the *visiblePoints* and *visibleNodes* variables keep track of the amount of points and nodes that were found to be visible. Traversal stops if there are no more nodes to visit or if a user-defined point budget has been reached.

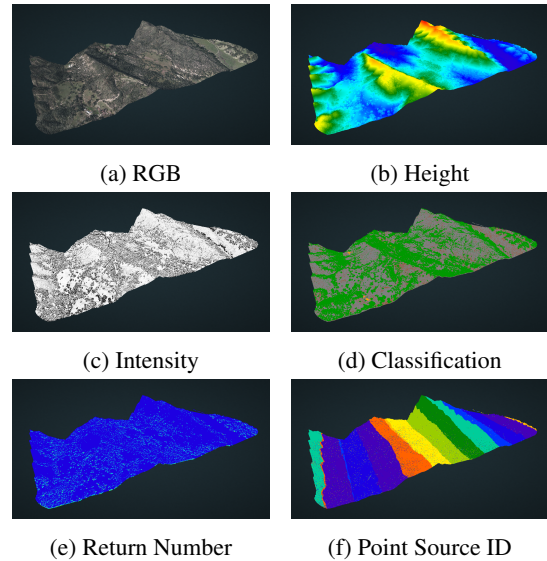


Figure 1: Coloring different point cloud attributes; CA13 point cloud courtesy of [8]

Frustum culling is done using box frustum intersection tests. A node that is not inside or does not intersect the view frustum will not be rendered and is omitted from further processing.

Nodes with a small projected size have a smaller impact on quality. They are discarded if their size is lower than a user-defined threshold.

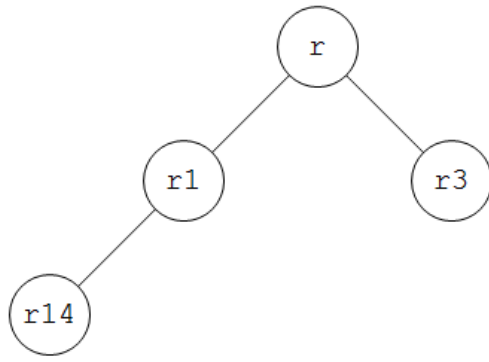
The visible hierarchy is kept in a list for use with other features such as ray casting, point picking and adaptive point size, which is explained in detail in subsection 6.3.

6.2 Coloring

Point clouds can have a variety of different attributes. LIDAR data, for example, often contains intensity, return number, point source ID and classification, but not necessarily color (RGB). Photogrammetry-based point clouds, on the other hand, have color and possibly normals, but no intensity or return number. In our system, octree nodes are stored as files on the disk, and the format of these files can be chosen freely. For point clouds with only color data, points are stored in binary files with coordinates and color. For point clouds with color, intensity, classification or return number, points are stored in LAS or compressed LAZ files, instead. Color data is rendered as is, while intensity is rendered using a grayscale gradient, classification is rendered using a look up table, and other attributes are rendered using rainbow color gradients, as shown in Figure 1.

6.3 Point Size

One problem of level of detail using multi-resolution hierarchies is the noticeable difference in point density in



ID	index	children	offset
r	0	00001010	1
r1	1	00010000	2
r3	2	00000000	0
r14	3	00000000	0

Figure 2: Hierarchy encoded in a 1D array containing children masks and relative offsets to a node’s first child.

regions with different level of detail. To overcome this problem, an adaptive point-size mode was implemented. This mode sets the point size based on both, the visible octree depth at the point position and the distance to the camera. Calculating the visible octree depth for all visible points and sending this data to the GPU each frame is a relatively slow process. Instead, this calculation is done directly on the GPU. The visible hierarchy is encoded in a 1D array, stored in a texture and sent to the GPU. Inside the vertex shader, the hierarchy is traversed towards the vertex position and the *localOctreeDepth* variable is incremented along the way.

Nodes are stored in breadth-first order and are encoded in 3 bytes, each. The bits of the first byte indicate which children are visible. The second byte contains the relative offset to the node’s first child inside the array. These 2 properties are sufficient to traverse the octree from top to bottom. The third byte is empty. Figure 2 shows an example of a hierarchy and its encoding. Each traversed level requires one texture lookup, counting how many bits have been set in a byte and whether a certain bit has been set or not. This puts additional overhead on the vertex shader, but it also reduces the number of vertices and fragments required to fill holes.

The point-size is calculated by taking the spacing of the octree root, then halving the size for each visible node at that location and finally computing the screen projected size, as shown in Equation 2 and 3.

$$worldSpaceSize = \frac{spacing}{2^{localOctreeDepth}} \quad (2)$$

$$pointSize = project(worldSpaceSize) \quad (3)$$

This algorithm only works properly if the *worldSpaceSize* is higher than the sampling density of the original point cloud data. In regions where the size is lower than the original sampling density, holes will appear because there are not enough points available to make up for the decreased point size. For a good utilization of adaptive point size it is therefore important that the octree depth is not too high.

Figure 3 shows the difference between fixed and adaptive point size.

6.4 Rendering Quality

Most point cloud viewers render points either as squares or circles. Increasing the size will cause these primitives to overlap and reduces the readability of high-frequency features such as text and fine details. In order to improve readability, the high-quality splatting [10] algorithm using screen-space aligned circles was implemented. Instead of rendering only the fragments closest to the camera, fragments within a certain distance are blended together. This algorithm requires at least 3 rendering passes. In the *depth pass*, a linear depth map with an additional linear offset, for example 1cm, is rendered. In the next step, the *attribute pass*, all fragments that pass the depth test, i.e., all fragments closest to the camera as well as fragments at most 1cm behind them, will be blended together. The fragments are weighted according to their distance to the center of the point, and the weighted value as well as the weight will be summed up. The last step is to normalize the buffers by dividing the weighted sum of the attributes by the sum of their weights. Both, blend depth and weight function can be made dynamic to adapt to different needs. Large-scale point clouds will need a different blend depth than point clouds of small objects. Changing the weight function can result in very smooth or blurry but also very sharp images. High-quality splatting gives very good results, but the downside is the need for at least 3 rendering passes.

In order to achieve good results in just one pass, we implemented another point-rendering algorithm with a similar idea. High-quality splatting assigns weights to fragments based on their distance to the point center and then blends them together. Instead of summing up weights and attributes, we use a weight function as an offset to the fragment depth. Essentially, this means that instead of rendering screen-aligned squares or circles, each point is rendered as a three-dimensional object. Fragments far from a point’s center are more likely to be occluded due to their high depth offset. The result is a nearest-neighbour-like interpolation with similarities to a Voronoi diagram.

Figure 4 shows a comparison of the different rendering modes.

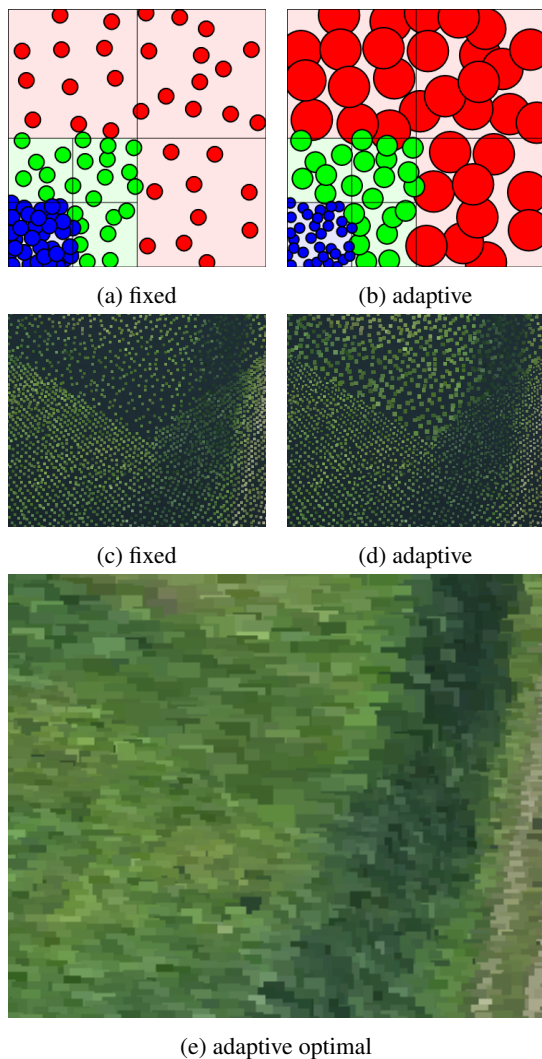


Figure 3: With fixed or projected point size, noticeable holes appear in regions with a lower level of detail. In figure (b) and (d), adaptive mode is used to increase point size in order to avoid holes. In figure (e), point size was chosen to avoid holes completely. Highway construction point cloud courtesy of [9]

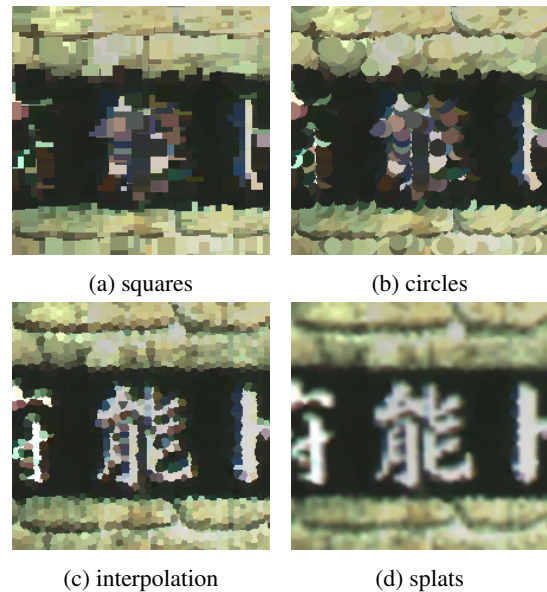


Figure 4: 4 different rendering modes. Squares and circles have issues with overlapping points. Interpolation and splats provide considerable improvements in readability of high frequency features. Point cloud courtesy of [11]

7 Georeferencing

One of the largest use cases of point clouds is the capturing of landscapes with LIDAR or photogrammetry. These kinds of point cloud scans can be georeferenced, which means they can be assigned coordinates that refer to exact positions on the planet. A variety of projections exists for different tasks and locations. These projections are usually planar and do not account for the earth's curvature, but they work as a good approximation in small regions. Additionally, coordinates are often stored as 32bit floats or integers which have enough precision for a given area but not the whole planet. Most projections are therefore only valid for small regions or countries and not meant to be used outside.

One of the challenges of working with georeferenced data are the huge values of point coordinates. According to Spatial Reference [12], x coordinates in EPSG:21781 (Swiss) projection lie between 485869.5728 and 837076.5648. Floating-point types have high precision for values near 0 but cannot accurately handle such high values. For this reason, the point cloud will be transformed to local scene coordinates by translating all points to the origin.

We use georeferencing to provide a side-by-side view of a 3D point cloud scene and a web map such as OpenStreetMap [13], as shown in Figure 5. In order to display camera position and direction in a web map, the camera scene coordinates are first transformed to the projected point cloud coordinates and then to the map coordinates. The reverse can be done as well, for example to draw a



Figure 5: Map overlay showing camera position and point cloud extent. Point cloud courtesy of [9]

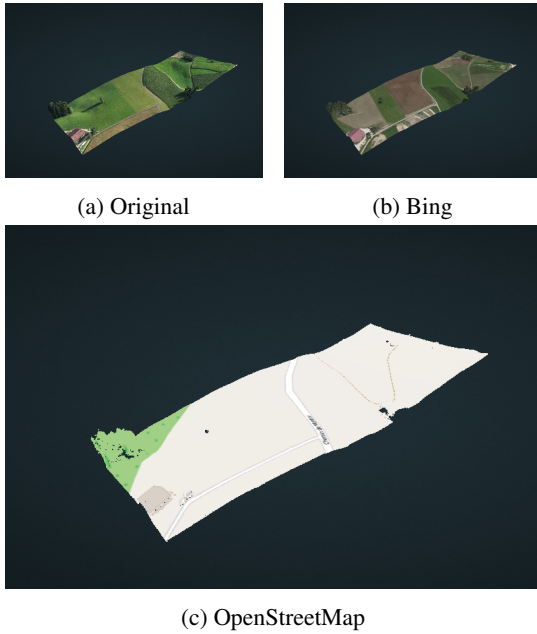


Figure 6: Bing and OpenStreetMap projected on point cloud data. Point cloud courtesy of [9]

line on the map and then display the same line in the 3D scene.

We also use georeferencing to project web maps, e.g., OpenStreetMap or Bing Aerial [14], onto point cloud data, as shown in Figure 6. First, the bounds of the desired region are calculated and then transformed to map coordinates. The map inside these coordinate bounds is used as a texture that is projected onto the point cloud data. Map projections may be used for colorless LIDAR data or to compare map data with point cloud data.

8 Performance

Indexing performance is measured in points per second. Unlike other methods, the presented method also subsamples the point cloud, so another important measure is the sampling ratio of points read versus points written. Table 1

Dataset	spacing	depth	points/s	ratio (%)
CA13	315	10	33k	49
		9	95k	14
		8	121k	4
		7	178k	1
Lion	0.05	4	98k	77
		3	133k	29
		0	333k	1.8

Table 1: Indexing Performance. points/s gives the number of points per second that were processed and ratio the amount of points that were written. Close points are discarded. All tests were done on the same 5400 rpm hard drive.

GPU	points	mode	FPS
860M	2M	fixed	104
	2M	adaptive	100
330M	1M	fixed	14
	1M	adaptive	11

Table 3: Performance of fixed and adaptive point-size modes. In both cases, 1 pixel was rendered per vertex to avoid influence of the fragment shader.

shows our performance results.

Rendering performance on 2 different notebooks is shown in Table 2.

Performance of adaptive point-size is shown in Table 3.

9 Conclusions and Future Work

We have shown a method to render large amounts of point cloud data in real time in web browsers without the need to download the full dataset. A pre-indexing step is required to sort points into an octree structure, which makes it possible to efficiently load just small parts of the dataset needed for the current view point. For georeferenced point clouds, we also showed how to synchronize them to web maps such as Bing or OpenStreetMap in order to display a map overlay showing current camera position and direction and also how the map can be projected onto point datasets.

Figure 7 shows some screenshots of point clouds that were rendered with our work.

Some important future tasks include improved indexing performance, supporting normals for illumination and avoiding generating a large amount of nodes with a small amount of points in each. The biggest problem with indexing right now is that a relatively slow dart-throwing-type algorithm is used in order to guarantee a minimum distance between points and to sample evenly distributed points without regular repeating patterns.

Additionally, further improvement is needed to make the adaptive point-size mode work with any octree depth

Dataset	#points	#rendered points	#rendered nodes	GT 330M(fps)	GTX 860M(fps)	Adreno 320(fps)
CA13	5500M	400k	77	41	148	12
		991k	115	21	122	4
		1987k	198	10	97	
Lion Statue	0.34M	150k	64	60	136	36
		340k	199	24	93	18
Matterhorn	90M	986k	81	11	120	8

Table 2: Rendering Performance Results. All tests were done in Chrome on 2 notebooks and a mobile phone: A Sony VPCF11C4E(2010) with a Nvidia GT 330M, a custom-built Schenker(2014) with a Nvidia GTX 860M and a Samsung Galaxy S4 Active with a Adreno 320. Tests were done with a point budget of either 1 or 2 million points. We have measured the frames per second (FPS) for different view points and listed the number of points and nodes that were rendered. For all measurements, adaptive point size was used to cover holes. For the notebook tests, a 1920x943 pixel canvas was used.

and to convert the whole dataset without discarding points that are close together.

10 Acknowledgements

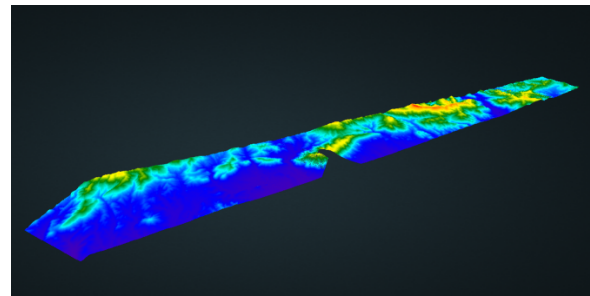
The CA13 dataset with a total of 17.7 billion points was taken from OpenTopography [8]. The highway in construction dataset is courtesy of Sigeom SA [9]. The funou dataset used for the point quality section was released online by Anan [11]. The matterhorn point cloud is courtesy of Pix4D [15]. This research was supported by the EU FP7 project HARVEST4D (no. 323567).

References

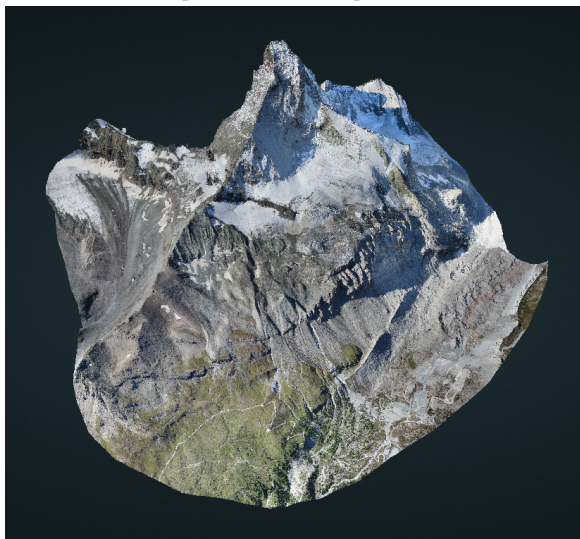
- [1] Michael Wimmer and Claus Scheiblauer. Instant points: Fast rendering of unprocessed point clouds. *Proceedings Symposium on Point-Based Graphics 2006*, 2006.
- [2] Markus Schütz. Potree. <http://potree.org>. Accessed: 2015-02-13.
- [3] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. *SIGGRAPH '00 Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 2000.
- [4] Enrico Gobbetti and Fabio Marton. Layered point clouds. *Computers and Graphics archive Volume 28 Issue 6*, 2004.
- [5] Howard Butler and Uday Verma. Plasio point cloud viewer. <http://plas.io/>. Accessed: 2015-02-13.
- [6] glob3mobile inc. glob3mobile framework. <http://www.glob3mobile.com/>. Accessed: 2015-02-13.
- [7] Sharelidar laser scanning sharing platform. <http://www.sharelidar.com/>. Accessed: 2015-02-13.
- [8] Opentopography. <http://www.opentopography.org/>. Accessed: 2015-02-14.
- [9] sigeom sa. <http://www.sigeom.ch/>. Accessed: 2015-02-14.
- [10] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on todays gpus. *Eurographics Symposium on Point-Based Graphics*, 2005.
- [11] Anan survey. <http://anan.skr.jp/>. Accessed: 2015-02-14.
- [12] Spatial reference. <http://spatialreference.org/>. Accessed: 2015-02-13.
- [13] Openstreetmap. <http://www.openstreetmap.org/>. Accessed: 2015-02-14.
- [14] Bing maps. <http://www.bing.com/maps>. Accessed: 2015-02-14.
- [15] Pix4d. <https://pix4d.com/>. Accessed: 2015-02-14.



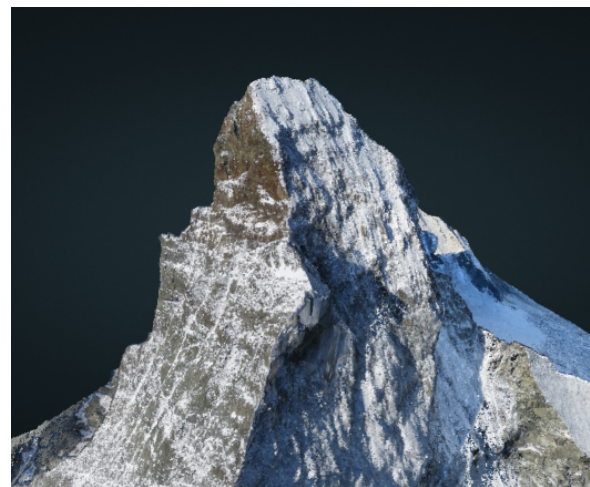
(a) CA13 point cloud 1.4M points rendered



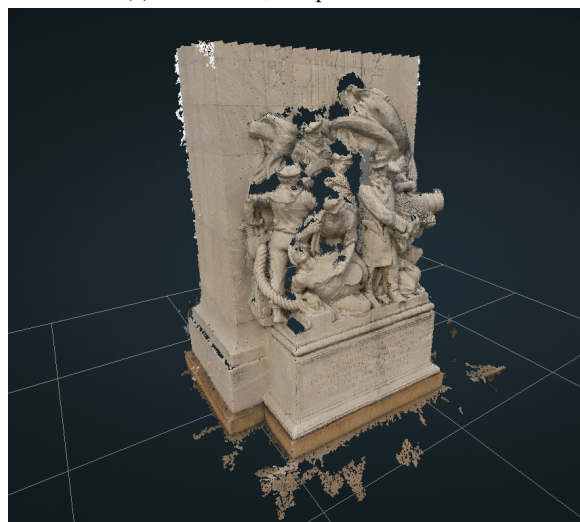
(b) CA13 point cloud 1.1M points rendered



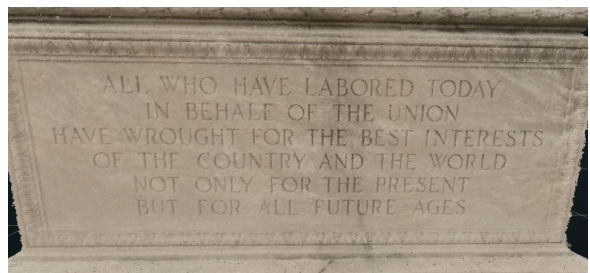
(c) Matterhorn, 1M points rendered



(d) Matterhorn summit, 1M points rendered



(e) Statue in Philadelphia, PA, 1M points rendered



(f) Closeup of statue inscription, 1M points rendered with interpolation shader

Figure 7: Screenshots of different point clouds, rendered either in Firefox or Chrome. CA13 point cloud courtesy of [8]. Matterhorn point cloud courtesy of [15]

Order Independent Transparency with Non-local Opacity Modulation for 3D Meshes

Tomáš Pastýřík*

Supervised by: Ladislav Čmolík†

Department of Computer Graphics and Interaction
Czech Technical University in Prague, Czech Republic

Abstract

We are rendering semi-transparent 3D objects on GPU we can choose from a variety of order independent transparency (OIT) algorithms. The transparency of the objects can be modulated based on properties of the 3D objects such as curvature, distance from silhouette, distance from camera, etc. In this paper we focus on non-local opacity modulation where desired information needed for the modulation is a matter of global context and it is not known for current primitive directly. We introduce an algorithm to solve the Order Independent Transparency with non-local opacity modulation based on the Illustration Buffer. While the original Illustration Buffer is constructed from meshes of flow surfaces we focus on use with general 3D meshes.

We compare our algorithm with several OIT algorithms: depth peeling, dual depth peeling, and per pixel linked lists which provides us a deeper insight at what conditions is one algorithm better than another from the point of speed, memory consumption and effort needed to incorporate the transparency modulation based on a certain property of the 3D objects to the algorithm.

Keywords: order independent transparency, non-local opacity modulation, illustration buffer, comparison, depth peeling, dual depth peeling, per pixel linked lists, opengl

1 Introduction

We are rendering semitransparent 3D objects, the order of the rendered primitives is critical to correctly compute the final colour of each pixel. Considering only objects consisting of 3D meshes it is not trivial to determine the order of the primitives, e.g. triangles, given by distance from the camera. A group of methods that do not require the meshes to be sorted before the rendering process is called the *Order Independent Transparency (OIT)*.

When the general *OIT* problem is solved we can also consider opacity modulation techniques to enhance perception of object's inner structure. We can classify such

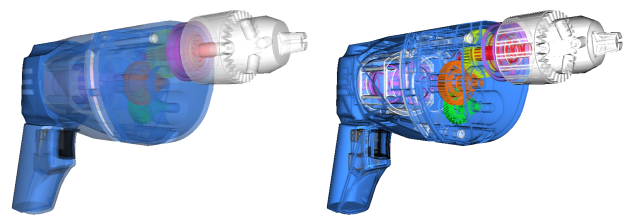


Figure 1: On the left all fragments are set 40% opacity. Non local opacity modulation (distance from silhouettes) is used on the right image to reveal inner structure of rendered drill.

techniques considering the knowledge of the information desired for the modulation as follows: *Local*, if information is known to a primitive - in case of this paper to a fragment - directly. This includes techniques based on the fragment lighting and shading, distance from the camera, or custom fragment properties. *Non-Local* is a complementary class to a local opacity modulation. It can be further divided to following cases: a) the information is retrievable from the direct neighbours along the surface or along the view ray. This e.g. includes modulation based on surface curvature or edges detection. b) the information is further than in case a). This e.g. includes modulation based on distance from important object features like silhouettes, etc.

Non-local opacity modulation is often needed to enable the user to see the required detail without losing the context. In this paper we introduce an OIT algorithm capable of both local and non-local opacity modulation. The algorithm is based on Illustration Buffer proposed by Carnecky et al. [3]. While they construct the Illustration Buffer from 3D meshes of the flow surfaces only our algorithm is designed for general 3D meshes. The main contributions of this paper include:

- The proposed OIT algorithm for 3D meshes along with the measurements of the algorithm stages and stages variants that provide better insight to the behaviour and performance bottlenecks of the algorithm.
- Comparison with existing algorithms solving OIT w.r.t. the speed, memory requirements, and ease of use.
- Our comparison also allows to decide which algorithm to use w.r.t. chosen the opacity modulation technique.

*mail@tomaspastyrík.cz

†cmolikl@fel.cvut.cz

2 State of the Art

Problem of the Order Independent Transparency (OIT) is well known in 3D scene rendering and there is no standard implementation included in either OpenGL or DirectX. It is a general problem consisting of rendering objects with a uniform or non-uniform alpha channel. To display such geometry correctly all fragments need to be blended in the correct order, thus sorting the fragments is often the key requirement for techniques solving OIT. In this section we briefly review algorithms that solve OIT using the rasterization proces. Please note that there are also *alpha blending approximations* [6][2] that perform approximative rendering in only one pass. Even though these approach are very fast, they only approximate the OIT problem and are not extendable to any methods considering non-local transparency and therefore these methods are not further examined in this text.

Depth Peeling presented in 2001 by C.Everitt [5] is based on multiple geometry passes, peeling just one layer of visible geometry per pass. It is in fact based on a shadow mapping technique, which helps to determine visibility between scene points and a certain light source. This algorithm process the scene by layers peeling one by one using two depth buffers per geometry pass.

While more advanced algorithms such as Dual Depth Peeling [2] blend these layers “on the fly” during the peeling passes, depth peeling algorithm [5] stores currently retrieved layer and performs another blending pass using full-screen quad, using OpenGL blending functions.

Dual Depth Peeling method by Bavoil [2] is a modification of the original Depth peeling algorithm allowing to peel two layers at once. In one pass it peels back and front layers simultaneously. Since this is not possible to do with the default depth buffer and GPU does not have multiple depth buffers to perform front to back and back to front rendering, custom min-max depth buffer has to be used.

To prevent peeling any fragments by both *front to back* and *back to front* directions, the algorithm uses mechanism of sliding window for two consecutive layers. While in the original depth peeling N geometry passes are necessary to process the scene, where N is the number of layers it created, Dual depth peeling performs $N/2 + 1$ geometry passes only. This algorithm however speeds up the rendering only if application is geometry (vertex) bounded.

Per Pixel Concurrent Linked Lists Another method is to store every fragment that belongs to one pixel in a linked list and sorting it by fragment’s depth to determine the order of the fragments. Method [9] described below is very similar to *A-buffer* [4], it only achieves OIT by using linked lists constructed in memory of GPU. While the first GPU implementations of A-buffer presented by Meyers and Bavoil A-buffer [7] and Bavoil et al. [1] were able to store fixed amount of fragments per list, the method presented by Yang [9] is unbounded.

A GPU version of A-buffer can be constructed in two rendering passes. In the first pass we create a linked lists of

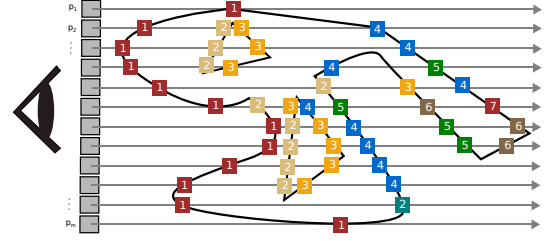


Figure 2: For the sake of simplicity *one row display* of pixels $p_i, i \in \{1, \dots, m\}$ where m is the number of pixels is shown. Fragments with the same number are in the case of *peeling* methods in the same layer. In case of *ray casting* terminology, numbers denote the order along the ray.

fragments per pixel by rendering the geometry and storing all fragments of a pixel to the linked list. The linked list can be accessed by an index to the first fragment called the *pixel head*. The second pass consists of a full-screen quad rendering and sorting the linked lists. Traversing the sorted linked lists to compose final pixel color can be done at the end of the sorting pass thus no further passes are needed.

Illustration Buffer was presented by Carnecky et al. [3] and inspired by Yang [9]. The Illustration buffer data structure is motivated by several image enhancements that modulate opacity based on non-local information.

To provide the information about the surrounding shape of all fragments, A-buffer constructed in the GPU memory [9] is extended. While in A-buffer method fragments know their neighbours only along the viewing ray, Carnecky et al. present methods to find and connect also neighbours that belong to the surrounding pixels. For pixel with coordinates (x, y) new four neighbours are found in linked lists of pixels $(s + dx, y + dy)$ where $(dx, dy) \in \{(1, 0), (-1, 0), (0, 1), (0, -1)\}$. After the neighbours are found the Illustration buffer can be used to traverse object surfaces to retrieve information about their shape, such as gradients, or distances to important features.

Structures created by our algorithm described in the next section are the same as used by Carnecky et al. [3]. We take advantage of the indexed geometry and propose a geometry motivated method to locate the neighbours which is faster and more precise than heuristics used by Carnecky et al. [3].

3 Proposed Algorithm

In this section we describe the Illustration buffer as well as our extension of the approach. The Illustration buffer requires the per pixel concurrent linked lists to be created first. However, in contrast to the *concurrent linked lists* we need to store much more information. In the following list are the buffers we need to construct and work with the Illustration buffer:

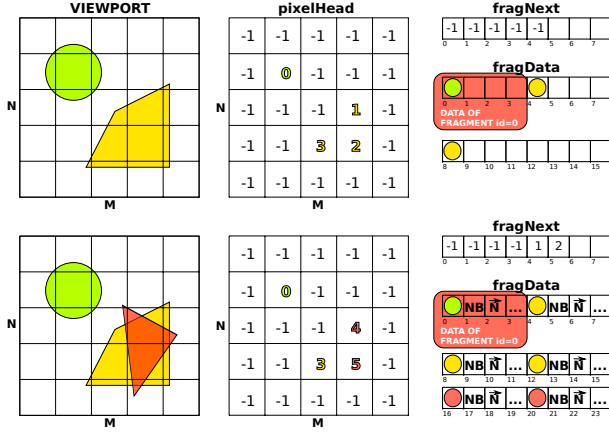


Figure 3: Shows the filling step of the algorithm. Data are spanned by four, giving a space for colour, four surrounding neighbours (NB), normal (N) and some other data, which depend on the target use of the buffer. Buffer pixelHead if of a same size as the viewport ($X \times Y$). This Figure does not consider the sorting of samples along the ray or search for the neighbouring fragments.

- **pixelHead** is a buffer of size $X \times Y$ where X, Y are the dimensions of the viewport. It stores ID of the first fragment in the linked list.
- **pixelCount** is also $X \times Y$ buffer storing lengths of the lists in each pixel. While not required when traversing the list and using special value for the end, this value is important for certain opacity modulation techniques.
- **fragNext** is a one dimensional buffer where the next index is stored for current fragment
- **fragData** stores all data we need to work with the Illustration buffer. For each fragment it stores the colour, indices of its four geodesic neighbours and optionally other data we need for non local transparency.
- **fragData2** is of the same layout as *fragData* and it is used for ping pong computational schemes
- **fragElements** stores 3 indices of the originating triangle per fragment, is used in our neighbours search

Figure 3 shows necessary structures for the *Illustration buffer* and how the data are stored when new element is rendered. To retrieve the next available free index for inserted fragment we need to use the global atomic counter `GL_ATOMIC_COUNTER_BUFFER`. We need a memory to store the indices of the fragment neighbours as well as custom fragment properties. Therefore we reserve several cells in the *fragData* buffer per fragment and use the spanning mechanism for retrieving the index to this buffer. The internal formats used by those buffers are: `GL_RGBA32UI` for the *fragData*, *fragData2* and *fragElements* buffers and `GL_R32I` for *fragNext*. The buffer *fragElements* consists of the vertex indices of the triangle it belongs to. The buffer *pixelHead* which stores pointers to first node of each per pixel linked lists is defined as `GL_TEXTURE_2D` with `GL_R32I` type of the viewport size.

3.1 Filling Step

Algorithm 1 shows the filling step of the algorithm when the geometry is rendered. Since we need to not only read but also write to the buffers in the later shader invocations, traditional `GL_TEXTURE_2D` cannot be used to store the data. Therefore we use the *ARB_shader_image_load_store* OpenGL extension. This extension brings functions *imageLoad()*, *imageStore()* and also many atomic operations *imageAtomic**(***). Packing of 4 floats $f, f \in [0, 1]$ to one unsigned integer using bitwise shifts and GLSL built-in functions *floatBitsToUint* and *uintBitsToFloat* is used to reduce amount of used memory. In the beginning of each frame we reset the atomic counter and buffer *fragData* using `GL_PIXEL_UNPACK_BUFFER` to initial state. The lines 10 and 11 of Algorithm 1 have to be atomic to prevent read-write collisions.

3.2 Sorting

In our application two sorting methods are implemented. The first is sorting the linked list without using any auxiliary structures and in the second is used an array of fixed size for sorting. Both presented methods are invoked simply by rendering a full-screen quad with access to the algorithm buffers storing the linked lists.

Sorting the Linked List by Insertion sort can be done easily using two *fragNext* buffers. One to be filled initially and second that will be used for adding sorted fragments as shown in Algorithm 2. We can see this procedure as an analogy to two linked lists A, B. A is unsorted and B consists only from copy of head in A. Then we remove node *a* from the front of A and insert it to B. To be able to remove *a.next* from A and insert it to B, we would have to remember what was the original *a.next* since inserting the node *a* to B may change its next pointer. In single linked lists this could be solved also by copies of the nodes instead of their

Algorithm 1: Filling per pixel linked lists

Data: Geometry to be rendered, atomic counter $AC = 1$
Result: Unsorted concurrent linked lists

```

1 Render the geometry, Depth Test OFF
2 while fragments with (X,Y) to be processed do
3     index = AC
4     colour = shadeFragment()
5     if pixelHead(X,Y) == -1 then
6         pixelHead(X,Y) = index
7         fragData(index*span) = colour
8         fragNext(index*span) = -1 //next pointer is empty.
9     else
10        nextPointIndex = pixelHead(X,Y)
11        pixelHead(X,Y) = index
12        fragData(index*span) = colour
13        fragNext(index*span) = nextPointIndex
14    end
15    AC += 1 // increase the atomic counter
16    discard fragment // we do not want it to be seen yet.
17 end

```

Algorithm 2: sorting the linked lists directly

Data: Buffers `u_fragNext` and `u_fragNext2`, `u_fragData` and `u_pixelHead`
Result: Sorted next pointers in the `u_fragNext2` and a head pointer in `u_pixelHead`.

```

1 int sortedSize = 1, int head = loadHead(x,y)
2 int newFrag = next(head); int current, int previous, float currentDepth
3 while sortedSize < totalCount do
4     if newFrag.depth < head.depth then
5         newFrag.nextSorted = head
6         head = new, head.depth = newFrag.depth
7         new = next(newFrag)
8         sortedSize++, continue
9     end
10    previous = head
11    current = head.nextSorted, int innerCounter = 0
12    while innerCounter <= sortedSize && current.depth < new.depth do
13        previous = current
14        current = current.nextSorted;
15        innerCounter++;
16    end
17    newFrag = new.next; sortedSize++;
18 end

```

3.3 Neighbours Location by Carnecky et al.

However wanted neighbours differ greatly from the neighbours in the layers. Let us consider situation depicted in Figure 4 where we see the found neighbours and peeled layers. It shows the difference between *neighbours of P in peeled layer* and *neighbours of P on the surface* discussed later.

Figure 4: *Left*: Using a perspective we show two crossing planes a). When peeling the first layer and querying the neighbouring pixels of P in such layer, we retrieve b). When searching for neighbours of P in Illustrative buffer we want to find c). *Right*: Geometrical meaning of ϵ_n and ϵ_z on the surface samples.

VERTICES COORDINATES

V_0	V_1	V_2	V_3	V_4	V_5						
0	1	2	3	4	5						

a)

b)

c)

d)

Figure 4. They compute the ε_n as difference of fragments i, j normals n_i, n_j as:

The eye distance ε_z is computed using the radius of a rendered object bounding sphere r_{obj} , normal n_i , pixel coordinates x_i , eye distance z coordinate and finally the z_i gradient $\left(\frac{dz_i}{dx_i}\right)$:

$$\mathcal{E}(i, j) = w_z \cdot \mathcal{E}_z(i, j) + w_n \cdot \mathcal{E}_n(i, j)$$

Proceedings of CESC G 2015: The 19th Central European Seminar on Computer Graphics (non-peer-reviewed)

3.4 Proposed Neighbours location

To overcome the inefficiency of the method [3] we propose a new method motivated by indexed geometry. Given two neighbouring lists A, B where list A is the current list and B is the list where neighbour is to be found, we propose auxiliary structure depicted in Figure 5.

We are using indexed geometry to lower the load of informations mapped to GPU memory. We extend the indices information so that every vertex knows indices of all vertices of the same triangle. This is shown in Figure 5 b). However this would be against the very principle of indexed geometry since we would replicate a lot of data. This can be solved as shown in Figure 5 c) where every triangle has its unique ID attached and auxiliary table to map ID s to triangle indices as shown in Figure 5 d).

For fragments $f \in A$ of coordinates x_f, y_f and $g \in B$ of coordinates x_g, y_g then apply following rules:

1. f and g are not neighbours if f and g do not share any indices of the triangle they are part of.
 2. f and g are neighbours and fragments of the same triangle if f and g share exactly 3 indices.
 3. f and g are neighbours and fragments of two neighbouring triangles if f and g share exactly 2 indices.
 4. f and g are neighbours and fragments of two neighbouring triangles if f and g share exactly 1 indices.
- This situation can happen e.g. for triangles with $ID = 1, ID = 4$ in figure 5.

The algorithm for neighbour search is then simplified to only one cycle through the neighbouring list B and there is no need for the cycle through A afterwards.

3.4.1 Drawbacks

Even-though this method is geometry motivated there can be artifacts caused by the rasterization process. Such artifacts occur when rendered triangles are smaller than pixel and neighbouring fragments skip triangle(s). This error is shown in figure 6. With that knowledge we can higher the viewport resolution or lower the detail of the model to overcome this.

3.5 Memory consumption

Unfortunately, memory consumption is the biggest weakness of the Illustration Buffer and therefore of our algorithm as well. While in Depth Peeling and Dual Depth Peeling structures are of fixed size without any relation to the number of rendered fragments (except for the absolute size of the viewport, of course), structures *fragData* and *fragNext* of the Illustration Buffer are growing linearly based on the number of fragments.

4 Results

In this section we present the results and measurements of our algorithm. We have implemented both presented

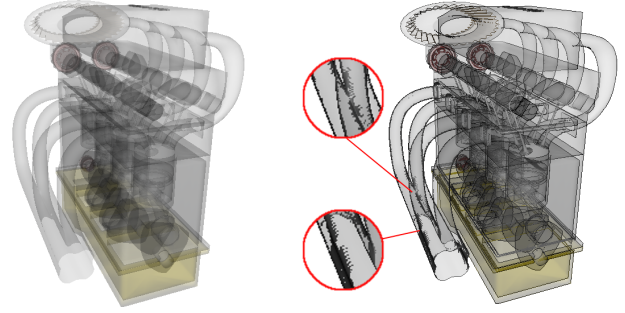


Figure 6: Every fragment has equal opacity in the left render of the engine. Right engine shows render in which fragment is fully opaque if the number of neighbours is less than four meaning it is part of the edge. Artifacts caused by the geometry detail and explained in the Section 3.4.1 are enlarged in the red circles.

Algorithm 3: proposed neighbour search

Data: two neighbouring lists A, B , current fragment $f_i \in A$, indices of f_i *indicesOfF*.

Result: Index to the linked list structure of f_i neighbour.

```

1 for  $b = 0; b < \text{count}(B); b++$  do
2    $\text{fragB} = B(b); \text{indicesOfB} = \text{fragB.triangleIndices};$ 
3   for  $k = 0; k < 3; k++$  do
4     for  $l = 0; l < 3; l++$  do
5       if  $\text{indicesOfB}[k] == \text{indicesOfA}[l]$  then
6         return  $\text{fragB.ID}$ ; // Neighbor has been found
7         since it shares at least one triangle index with  $f_i$ .
8       end
9     end
10  end

```

methods of the sorting and as the proposed neighbours search. Very precise OpenGL GPU queries are used to measure application rendering time in nanoseconds. For measurements of the total rendering time we use the `QElapsedTimer` from QT Framework. Twelve varied models are used to measure the Illustration buffer characteristics. Each model is tested in two positions - one general and one where the length of linked list is maximum possible. Note that in all measurements image resolution 600×600 is used if not stated otherwise. To see all the measurements conducted with all the tables and graphs, please see the master's thesis this paper originates from [8]. We have used GeForce GTX 660 with 2048 MB GDDR5 memory, $4 \times 2\text{GiB}$ DIMM DDR and Intel Core™2 Duo CPU E6850 @ 3.00GHz for all measurements. Final renders using different non-local opacity modulations are shown in Figure 16.

4.1 Sorting

The graph in Figure 7 is very clear that the dynamic version of the sort is winning in all cases over sorting in static array. For the measurements we have used 3 arrays of size 64. One for IDs, second for the depths and third for the distances between layers. In case of our GPU it was more expensive to allocate such arrays than much bigger amount of texture reads and writes, which could differ on

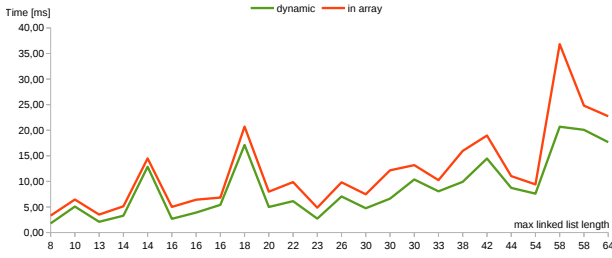


Figure 7: Speed comparison of the dynamic sort and sorting in static array. Bigger spikes are caused by the Ψ parameter which will be introduced when studying parameters that affect the performance of the algorithm the most.

hardware where invocation of the fragment shader for one pixel would have more memory available.

4.2 Illustration Buffer Performance

During the Illustration buffer creation we examine relations between number of vertices, rendered fragments, lengths of the linked lists storing the data along the ray, and rendering time on the GPU and rendering time combined with the CPU workload.

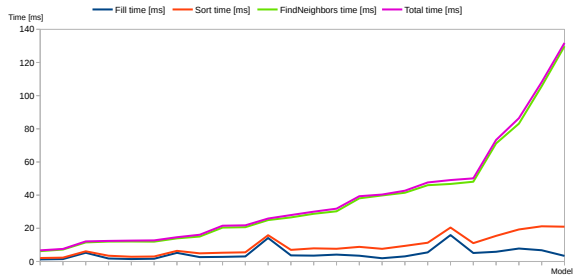


Figure 8: GPU Time in split stages of the Illustration buffer creation process. Models on the horizontal axis are sorted by the *Total* rendering time.

To be able to examine the relations fully, the creation process is split to several stages by the functionality. Figure 8 shows the performance of the separated algorithm stages. The time presented for each stage contain also times measured for all preceding stages. Figure 9 shows the relations between number of vertices, rendered fragments, lengths of the linked lists and their impact on the rendering time. We can see that another not mentioned parameter affects the rendering time significantly in Figure 9. Scenario that includes models GPU 2 and Suspension 2 is further shown in Figure 10.

Even though we process the scene on GPU, the process is not entirely parallel. All the parameters are higher for the Suspension 2 model than for GPU 2 model and yet the rendering time is greater for the GPU 2 model. The reason is that the number of long lists is much lower for the Suspension 2 model than for the GPU 2 model (see Figure 10). The dashed line in Figure 9 shows percentage coverage of the linked list lengths that are bigger than $\frac{2}{3}$ of

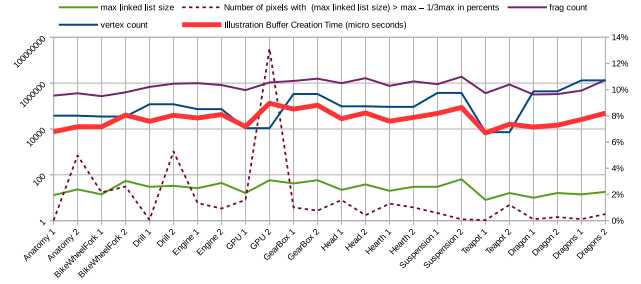


Figure 9: Secondary Y axis is used for the dashed line representing Ψ , primary Y axis (on the left) is then used for all other variables using logarithmic scale.

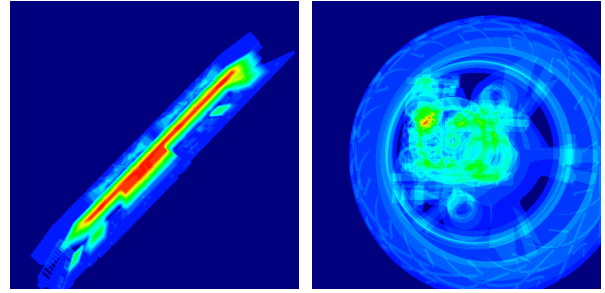


Figure 10: Heatmaps of the linked lists lengths of GPU 2 and Suspension 2. Color represents the distribution of the linked lists lengths, blue is zero and red is maximal linked list length.

the longest linked list, please mind the secondary vertical axis. We denote this parameter as complexity coverage Ψ .

4.3 Memory Consumption

Using the presented structures we need 208 bytes per fragment and additional 96 bits per pixel. This gives us for the scene Suspension 2 with 1900548 fragments and 600x600 resolution memory requirements of approximately 397MB. For 800x800 resolution it requires 702MB and for 1200x1200 we would need 1.58GB of GPU memory. Given this amount of data we are very likely to face an overflow of the used buffers on memory bounded systems.

5 Comparison of OIT Algorithms

This section provides comparison of the speed of our algorithm with the remaining methods: *depth peeling*, *dual depth peeling* and *concurrent per pixel linked lists*.

We have used implementation of the peeling methods from the NVIDIA Graphics SDK 10, only our own GPU time measuring system was added to their implementation. Concurrent per pixel linked lists are on the other hand measured using our own implementation since it is a sub-problem of the Illustration buffer construction. We have used GeForce GTX 660 with 2048 MB GDDR5 memory, 4 × 2GiB DIMM DDR and Intel Core™2 Duo CPU E6850 @ 3.00GHz.

Figures 11, 12, 13 and 14 show such comparison using parallel coordinates. In all presented graphs the render-

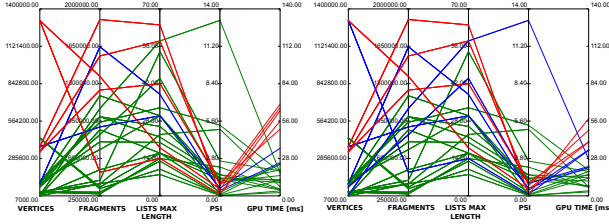


Figure 11:
Depth peeling

Figure 12:
Dual depth peeling

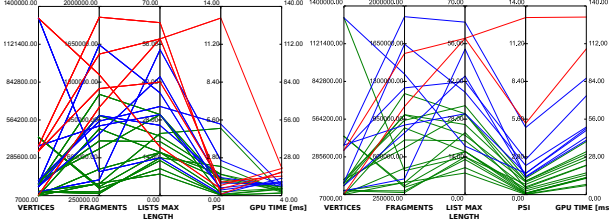


Figure 13:
Concurrent per pixel linked lists

Figure 14:
Illustration buffer

ing time is divided to thirds, where times in the first third are green, in second blue and times of the worst third are coloured red.

We can see that the *linked lists* absolutely win in speed. Another observation is that only the *Illustration buffer* is really affected by the complexity coverage Ψ , which is caused by its *FindNeighbours* stage. We can also see that the overhead on the fragment shader is not big for first three methods and they are vertex bounded in most cases. This is logical for the *peeling methods* since we need to render the geometry in each peeling pass. It might be however surprising for the *per pixel linked lists*, where even though the sorting procedure must occur on all fragments, the overhead is small thus application stays vertex bounded. However this is completely different in the case of the *Illustration buffer* where processing of the *FindNeighbours* stage is vital for the final rendering time.

During our measurements, the speed comparison of the *dual depth peeling* and the *depth peeling* we have observed that the *dual depth peeling* is in fact much slower than the *depth peeling* algorithm. The authors [2] admit that the *dual depth peeling* may speed up performance by 2x for geometry bound applications. This issue should be however stressed and explained much more in the original article. Reason for this behaviour is that the workload of fragment shaders is only worth the computation if the work of the vertex shader is more demanding which is a case of vertex bounded applications.

6 Opacity Modulation

Opacity modulation is a desired feature of the OIT solving algorithms. Therefore this section briefly demonstrates the power of the proposed algorithm.

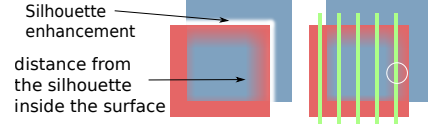


Figure 15: *Right*: Modulation by distance from the silhouettes, *Left*: Scene with the problematic situation in the white circle.

6.1 Local Opacity Modulation

When we are considering only the local opacity modulation using only information available to a fragment directly, all reviewed algorithms can be used. We only mention some of the techniques in this class for reader's convenience: based on the lighting intensity, custom per fragment data e.g. density, or by distance from defined plane/area (cut motivated). This class of modulation simply consists of all procedures that can be computed only from the global shader knowledge (uniform variables, constants, etc) and knowledge of the current fragment.

The *Depth peeling* is certainly the easiest one to be implemented but not the fastest. Dual depth peeling can speed up the rendering time only if application is vertex bounded as stated above. Implementing the per pixel linked lists is more challenging task but as we can see in the Section 5 it is certainly the fastest algorithm of all compared in this context.

6.2 Non-Local Opacity Modulation

The situation is much more challenging when the opacity of the current pixel depends on the context - on the state of its surroundings. While some opacity modulation techniques can be used quite easily using both peeling and linked lists mechanisms, some are not solvable by the peeling algorithms. Let us consider simple modulation by distance along the ray as in Equation 1. For two samples s with indices i, j where $i < j$ and therefore s_i closer to the camera, and user defined parameter *focusRegion* we compute opacity α of sample s_i as shown in Equation 1. This is achievable by simply comparing values of last peeled and current sample in the *peeling methods* and by traversing the sorted linked list easily.

$$\alpha = \text{saturate} \left(\frac{|s_i.\text{depth} - s_j.\text{depth}|}{\text{focusRegion}} \right) \quad (1)$$

Let us now consider a modulation by distance from important features of the 3D mesh, in this case the silhouettes. If we consider the problematic situation from the Figure 15, this situation is not simply solvable by the peeling methods since neighbours of the peeled layers are not always neighbours of the same surface. We therefore cannot know if the red gradient inside the circle should continue or not since we cannot tell if the surface is continuous under the green surface or if it exists at all. This situation is however solvable easily by proposed algorithm since we have knowledge of the surface neighbours on the surface of the mesh and not only of the neighbours on the current layer.

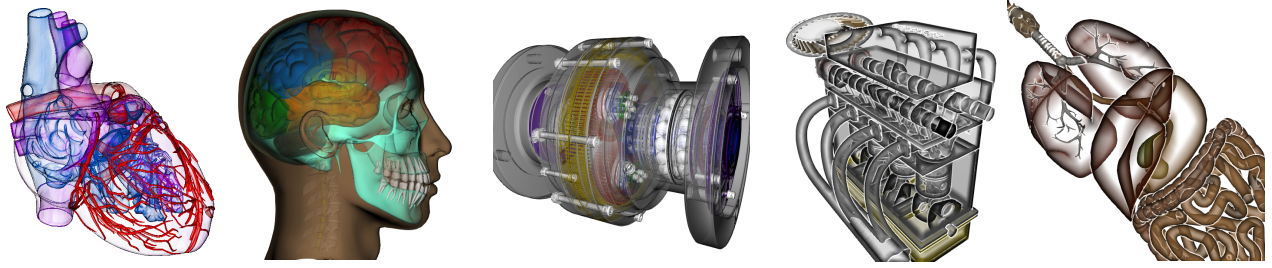


Figure 16: Final renders from our application using the Illustration buffer. These pictures were created by opacity modulation based on: surface curvature, distance between samples along the view ray, distance from surface silhouettes and silhouettes enhancements and their combinations.

7 Discussion of Results

The measurements of the OIT task alone shows clearly that methods based on the concurrent linked lists are much more efficient than the peeling methods. In case of the Illustration buffer the bottleneck is the neighbours location stage. To improve this process the future work should consider that in current form whole neighbouring list has to be traversed. We therefore suggest developing a heuristic based on the *interpolation search* optimized for the linked lists. If the approximate position of the desired neighbour is known, we can load only the next pointers to traverse to this node and omit loading its data.

Our measurements of the sorting methods shows that a bigger amount of read and write access to the OpenGL *image buffer* is more efficient than allocating an additional memory array for fragments to reduce number of *image buffer* accesses. This is a great motivation for developing dynamic algorithms that are unbounded. Proposed neighbours search method is faster and more precise than approach of Carneky et al. [3], but it can generate artifacts as discussed above. Speeding up the neighbours search process by interpolation search might give us enough time to examine neighbours lists of distance greater than one to eliminate these artifacts.

The complexity coverage parameter Ψ impact should be further researched - how it affects the algorithm parallel computation since it demonstrates the complexity of GPU parallelization process and optimizations being done by the hardware.

8 Conclusions

In this paper we present an algorithm to solve the Order Independent Transparency for general 3D meshes that allows non-local opacity modulation. This algorithm is based on the Illustration Buffer. We discuss two methods of the fragment sorting as well as measurements of both methods. Novel geometry motivated technique to find geodesic neighbours of fragments is proposed. Our method is faster and more precise than heuristic proposed by Carneky et al. [3]. However, further research is however required to eliminate presented artifacts.

To learn more about our algorithm we encourage the reader to read the thesis [8] this paper originates from. Figure 16 can serve as a motivation and demonstration of the Illustration buffer flexibility considering opacity modulation techniques.

The comparison of OIT algorithms provides insight into behaviour of the algorithms at different conditions and can be used to choose the right algorithm for the given conditions. While peeling methods are easier to implement, the best rendering times are achieved by implementing more complex per pixel linked lists.

References

- [1] Louis Bavoil, Steven P Callahan, Aaron Lefohn, João LD Comba, and Cláudio T Silva. Multi-fragment effects on the gpu using the k-buffer. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 97–104. ACM, 2007.
- [2] Louis Bavoil and Kevin Myers. Order independent transparency with dual depth peeling. Technical report, NVIDIA Corporation, 02 2008.
- [3] Robert Carneky, Raphael Fuchs, Stephanie Mehl, Yun Jang, and Ronald Peikert. Smart transparency for illustrative visualization of complex flow surfaces. *IEEE Trans. Vis. Comput. Graph.*, 19(5):838–851, 2013.
- [4] Loren Carpenter. The a -buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph.*, 18(3):103–108, January 1984.
- [5] Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 06 2001.
- [6] Houman Meshkin. Sort-independent alpha blending. 2007.
- [7] Kevin Myers and Louis Bavoil. Stencil routed a-buffer. In *ACM SIGGRAPH*, volume 7, 2007.
- [8] Tomáš Pastýřík. Visualization of inner structure of complex 3D objects based on opacity modulation. Master’s thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, Prague, Czech Republic, 2015.
- [9] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. In *Proceedings of the 21st Eurographics Conference on Rendering*, EGSR’10, pages 1297–1304, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

Vision & Perception

Fast Detection of the Pupil Centre in Stable Light Conditions

Krzysztof Wolski*

Supervised by: Radosław Mantiuk[†]

West Pomeranian University of Technology, Szczecin
Poland

Abstract

We present a simple and efficient technique of the pupil centre detection. This technique is addressed to the video eye tracking solutions, in which pupil centre must be found in image of the human eye. In contrary to previous work, we assume stable light conditions that provide a correct eye image. Such conditions can be achieved in many eye tracking applications but our solution is especially addressed to the scientific activities related to the perceptual experiments. We introduce a novel cross spread technique in which it is assumed that pupil shape is similar to ellipse. In this strategy, a parallel algorithm can be applied to detect the pupil centre, which enables accurate operation in less than 2 milliseconds. We present the OpenCL-based implementation of the cross spread algorithm and its application in the real-world eye tracker. The paper shows the results of the experimental measurement of this eye tracker accuracy performed for a number of human observers. The achieved accuracy close to 1.5 degree of the visual angle is comparable to the commercial devices.

Keywords: pupil centre detection, cross spread, eye tracking, eye tracking accuracy, perceptual experiments

1 Introduction

Detection of the pupil centre in the images of the eye is a basic function of the video-based eye trackers. This type of eye trackers consists of the infrared camera and the infrared light source, which are directed towards the eye. The camera captures the image of the eye with the dark circle of the pupil (see example in Fig. 2). The pupil follows the gaze direction during eye movement. Location of its centre is used to estimate the gaze direction.

The field of view for both eyes spans more than 180° horizontally and 130° vertically, although, humans are able to see details only in the fovea — the 2° patch of the retina located in the middle of the macula. The eye muscles enable fast gaze shifting to orient the eye such that the object of interest is projected onto the fovea. There are four types of eye movements: vergence movements, vestibular ocular movements, smooth pursuit, and sac-

cadic movements [6]. From the eye tracking perspective, the most important is the latter one. Fast (up to 900°/s) and short (10-100 milliseconds [1]) saccades move the eye to a new area of interest. They should be captured with at least 200 Hz frequency to allow accurate registration of the gaze direction. Otherwise, eye tracker can register a gaze location in the middle of the saccadic movement making its identification challenging.

In this paper we propose a novel pupil tracking algorithm called the *cross spread*. We assume that the pupil shape is similar to ellipse. Then, we use basic image processing operations to process the image of the eye and find the pupil centre. The algorithm consists of three steps: thresholding and binarization, noise reduction using median filter, and the core cross spread algorithm, which detects the pupil centre. These operations are sufficient to achieve the high accuracy of detection. However, a correct image of the eye must be delivered from the camera to avoid image analysis errors. The most important is a good visibility of the pupil on the iris background. A correct image of the eye can be taken in the stable light conditions. Such conditions may be provided e.g. during the perceptual experiments, that use eye trackers.

We implemented parallelised version of the cross spread algorithm based on the OpenCL library, which detects the pupil centre in less than 2 milliseconds. This implementation was tested with the Do-It-Yourself (DIY) eye tracker - the custom-built head-mounted eye tracking system [4]. The paper shows the results of the experimental measurement of DIY accuracy performed for a number of human observers. The achieved accuracy close to 1.5 degree of the visual angle is comparable to the commercial eye trackers.

In Sect. 2, an existing pupil detection techniques are outlined. The cross spread technique is introduced in Sect. 3, followed by the description of its parallel implementation and results of the performance tests. In Sect. 4 we present conducted experiments that measured the accuracy of the custom-build eye tracker equipped with our implementation of cross spread technique.

2 Previous work

Most algorithms for detection of the pupil centre binarise the image of the eye and filter out the noise to achieve the

*krwolski@wi.zut.edu.pl

[†]rmantiuk@wi.zut.edu.pl

best possible image of the pupil. Then, various scenarios are implemented to detect the centre of the ellipse, which shape reproduces the shape of the pupil.

The curvature detection algorithm [8] detects the edge of the pupil by tracing rays in all direction from the starting point located within the pupil area. For each ray, the location of the pupil edge is detected. Then, a heuristic curvature detection algorithm is used to eliminate the edge deteriorations caused by the eyelids, cilia, or corneal reflections. Finally, the parameters of ellipse, which best fits in the detected edge are computed using the least squares solution.

In the starburst algorithm [2] rather complex noise reduction technique based on Gaussian filtering, thresholding and morphological operation is used to achieve satisfactory image of eye. Then, similarly to the curvature detection technique, the rays are shot to find the pupil edge. Finally, the best fitting ellipse is determined using the RANSAC technique.

Other pupil centre detection algorithms are described in [7] and [5]. The goal of all these techniques is to achieve the best accuracy of detection. Much attention is paid to noise reduction and accurate ellipse fitting. In the following section we present simplified technique. We assume that correct image of the eye is delivered from the camera and more attention is paid to the processing speed. We propose the parallelised implementation of this algorithm adjusted to the GPU processing.

3 Cross spread technique

This section includes a detailed description of the cross spread algorithm and discussion on the drawbacks of this technique.

3.1 Detection pipeline

The whole algorithm consists of three steps presented in Fig. 1. As an input, the eye image is taken using the infrared camera. Then thresholding and median filtering is applied to binarise and denoise the image, respectively. Finally, the core algorithm is activated to detect the pupil centre and compute (x,y) coordinates of this point. The output values are expressed in pixels of the camera image.



Figure 1: The pupil centre detection pipeline.

The core cross spread algorithm is designed to search elliptic shapes. Any deviation of continuity of the edges or spatial coherence of the blob may reduce its accuracy. The possible deterioration depends mainly on the area of an artefact. Although the algorithm is designed to find the centre of ellipse, it can work correctly also for other convex shapes that are centrally symmetric.

Image thresholding

Thresholding is a simple image processing operation, which binarises the image colours. The infrared camera delivers image in the grey shades (see Fig. 2, left). After thresholding, the pupil ellipse becomes black and the rest of the pixels in an image become white. It is done based on the threshold value - empirically chosen grey value below and above which, the pixels are marked as black or white respectively. Fig. 2 (right) presents an example image of the eye after thresholding. The threshold value can be adjusted to the camera and lighting conditions. However, in some cases it must be tuned for an individual observer.



Figure 2: Left: image taken by the infrared camera. Right: the same image after thresholding.

Median filtering

The main task of median filter is to reduce the noise in the thresholded image. After thresholding, some black pixels can still exist in the pupil surrounding. These pixels are filtered out using the median filter. Filtration efficiency, i.e. the efficiency of impulse noise removal, depends on the size of this neighbourhood (see examples in Fig. 3). However, if one chooses too large surrounding the pupil shape can be distorted.

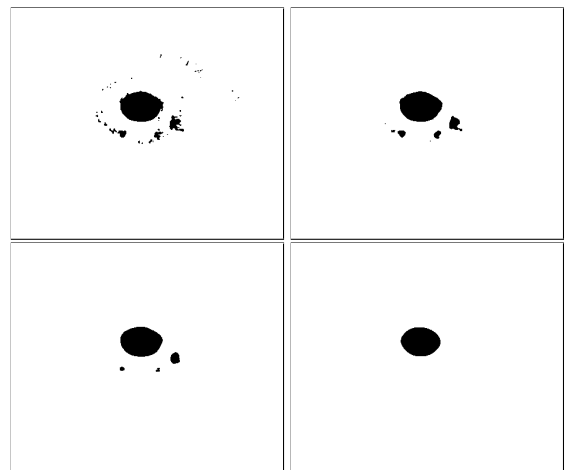


Figure 3: Results of the median filter for 3x3, 7x7, 15x15, and 31x31 pixel neighbourhoods (viewed from the top-left corner). The median filter removes pixels that not belong to the pupil.

The median filter is computationally complex operation for the colour images. It requires sorting of the pixels, which is particularly expensive for the large pixel neighbourhoods. For binary image, this operation is much faster, because the final value of the pixel depends on the number of white pixels in the surrounding. If this number is higher than the number of all pixels in the surrounding, the final value is set to 1 (white colour). Vice versa, the pixel value is set to 0 (black colour).

Cross spread algorithm

The term *cross spread* refers to the process of searching the boundary points. This technique is based on the human eye trait - circular shape of the pupil. Unlike pupils of other mammals, the human pupil is close to the circle. When the image of the eye is captured by camera, the pupil forms various elliptical shapes, depending on the location of the eyeball.

The cross spread technique is performed in the following steps:

1. Choose the starting point located in the area covered by the pupil (black pixels) (Fig. 4a).
2. Shoot 4 rays from the starting point in the horizontal (left, right) and vertical (top, down) directions (Fig. 4b).
3. Find the boundary points located on the pupil edge, that are closest to the starting point (Fig. 4c).
4. Create a new starting point by averaging the coordinates of the boundary points (Fig. 4d).
5. Iterate again from 1. until the location of the starting point is stabilised between iterations.

The boundary points are identified as points not belonging to the pupil, i.e. points with the high gradient between the current and the next position. Abscissa and ordinate of the new starting point are computed by averaging the coordinates of the horizontal and vertical boundary points, respectively.

Parallelisation

We implemented the parallelised version of the algorithm. We apply the regular grid of horizontal and vertical lines that covers the image. Candidate starting points are generated at the intersection of these lines. The ones belonging to the pupil are passed to further processing (see Fig. 5a). Then, four rays from each starting point are traced in parallel and the boundary points for each ray are located (see Fig. 5b). The location of the most advanced points in each direction is stored (see Fig. 5c). These values are used to find the pupil centre (see Fig. 5d).

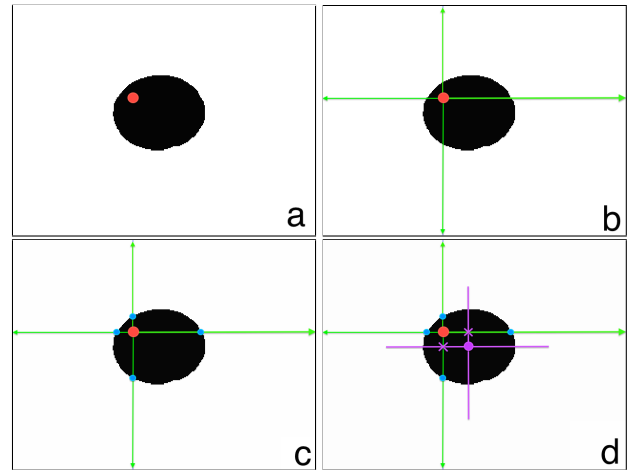


Figure 4: An iteration of the cross spread algorithm.

Corneal reflections

Corneal reflections are the small bright spots in the image of the eye (see Fig. 6). In the most eye tracking systems they are caused by infrared light sources placed near the camera.

If the corneal reflection is located within the pupil it interferes with the trivial implementation of the cross spread algorithm and can cause false detection of the boundary points. Therefore, after detecting a boundary point, we continue the search for the next few pixels. This tolerance depends on the size of the corneal reflection spots and should be set empirically.

3.2 Implementation and performance tests

We implemented the parallelised version of the algorithm using the OpenCL library¹. This library allows choosing the computing device, which can be both CPU or GPU. We created three kernels responsible for thresholding, filtering, and the core cross spread algorithm. The kernels share the same device memory.

We measured the execution time of the cross spread algorithm using the profiling system provided by the OpenCL platform. It enables to separate time spent for individual kernels and also measures the overall execution time. The test was run 1000 times and then, the results were averaged. The final results are presented in Tab. 1.

Four different computation devices were evaluated: Intel Core i7-2670QM (2.20 GHz), Intel Core i7-3537U (2.00 GHz), NVIDIA GeForce GT 555M, and NVIDIA GeForce GT 740M.

As it was expected, the best results were achieved for GPUs. Both graphics processors can compute the cross spread in less than 2 ms, which is equivalent to processing of more than 500 frames per second. The data preparation phase is rather expensive for GPUs (this is 70% of the

¹Open Computing Language, <https://www.khronos.org/opencl/>

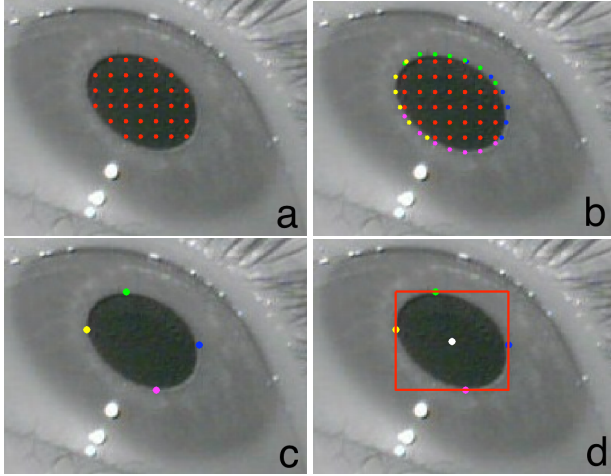


Figure 5: The parallel version of the cross spread algorithm. a: red dots depict the starting points. b: the blue, green, yellow, and pink dots are the boundary points. c: location of the extreme points. d: the white dot depicts the computed pupil centre surrounded by the red rectangle defining the boundaries of the pupil.

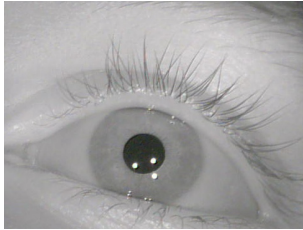


Figure 6: The two corneal reflection spots located with the pupil.

total execution time) because the camera image must be transferred to the GPU memory .

Interestingly, we achieved satisfactory results also for CPUs. The execution time of 4.4195 ms and 7.483 ms is equivalent to processing of 225 and 133 frames per second, respectively.

As can be seen in Tab. 1, the median filtering is the main bottleneck of the CPU implementation. It accounts for 60% of the overall execution time.

4 Accuracy evaluation

The goal of the experiment was to evaluate the accuracy of the cross spread algorithm. However, we measured this factor indirectly by testing the accuracy of the DIY eye tracker equipped with our software.

4.1 Do-It-Yourself eye tracker

The DIY eye tracker is a custom-built low-cost eye tracker of a basic construction [4] (see Fig. 7). It consists of two

Computing device	Intel Core i7-3537U 2.00 GHz	Intel Core i7-2670QM 2.20 GHz	NVIDIA GeForce GT 555M	NVIDIA GeForce GT 740M
Thresholding	0.9613	0.5080	0.0944	0.1388
Median filtering	4.4191	2.4243	0.1158	0.1776
Cross spread	0.3036	0.2001	0.2198	0.2128
Overall time	7.4830	4.4195	1.9031	1.8260
Data preparation	1.7989	1.2871	1.4731	1.2968

Table 1: Execution times of the paralleled version of the cross spread algorithm. Time in milliseconds.

main components: a modified safety goggles that act as a frame, and a typical web camera: Microsoft Lifecam VX-1000, working in 640x480 pixels resolution. The only change made to the camera is replacing the infrared light blocking filter with the visible light blocking filter to enable capturing images in the infrared light spectrum. The camera is mounted on the frame in 5 cm distance from the left eye. It is connected to computer via the USB cable. The eye is illuminated by three infrared LEDs placed close to the camera lens.

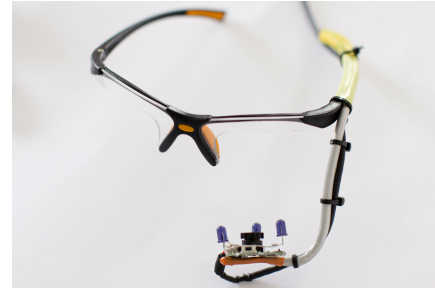


Figure 7: Head-mounted Do-It-Yourself video eye tracker [4].

Formerly, the DIY eye tracker was controlled by the ITU Gaze Tracker software. We reimplemented the whole eye tracking pipeline replacing all the tasks made by this software with our algorithms. Our implementation involves pupil detection based on the cross spread algorithm, but also communication with the camera, eye tracker calibration, and gaze position estimation.

The principle of the eye tracker operation is based on the observation that the pupil follows the gaze direction during eye movement. Therefore, the location of the pupil centre can be used to estimate the temporary gaze position/direction. The cross spread algorithm detects the pupil centre as the position in camera image coordinates (in pixels). These coordinates must be transformed from the camera space to the screen space to compute the gaze position on the screen. It is done using the mapping defined as the polynomial transformation citeRamauskas06:

$$\begin{cases} s_x = a_0 + a_1x + a_2y + a_3xy + a_4x^2 + a_5y^2, \\ s_y = b_0 + b_1x + b_2y + b_3xy + b_4x^2 + b_5y^2, \end{cases} \quad (1)$$

where (s_x, s_y) depicts gaze position in the screen coordinates (in pixels). $a_{0...5}$ and $b_{0...5}$ are coefficient computed during the eye tracker calibration. Calibration is the mandatory part which precedes every eye tracking session. During calibration, people are asked to look at the target points displayed on the screen, so one can assume that locations of these target points are known (i.e. (s_x, s_y) for each target point is known). The pupil centre (x, y) is computed by the cross spread algorithm in the camera space. Then, the polynomial coefficients can be calculated using the Singular Value Decomposition technique (SVD) [3]. During actual eye tracking, the polynomial with known coefficients can be used to transform the centre of pupil location from the camera space to the screen space.

Stimuli and procedure

Observers sat in the front of the display in 60 cm distance and used the chin-rest adopted from an ophthalmic slit lamp. The experiment started with a 9-point calibration. This procedure took about 20 seconds and involved observation of the markers displayed in different areas of the screen. The data processing including computation of the calibration polynomial coefficients was performed by the custom software.

In validation phase, participants looked at the circle marker displayed for 2 seconds at 25 different positions located on the regular grid (see Fig. 9). These positions, called the target points, acted as known and imposed fixation points. The marker was moved between target points in random order. We noticed that smooth animation of the marker between target points allows for faster observer's fixation and reduces number of outliers. Additionally, the marker was minified when reaches its target position to focus observer's attention on a smaller area. The data recorded before 800 ms from the beginning of the marker movement was removed from the analysis. Also the data collected over the last 200 ms were filtered out. Thanks to this it was possible to avoid the errors arising from the gaze transfer between the reference points.

The experiment was performed in a darkened room. Images were displayed on LCD monitor with native resolution of 1920 x 1080 pixels.

Participants

We repeated the experiment for 11 volunteer observers (age between 20 and 22 years, 9 males and 2 females). They declared normal or corrected to normal vision and correct colour vision. The participants were aware what they should do, but they were naïve about the purpose of the experiment.

4.2 Results

The accuracy of eye tracker is quantified as the average distance between the physical target position and the measured gaze position. During experiment described in Sect. 4 we registered over 30 thousand gaze points with the known reference. Due to the eye tracking inaccuracies these points are located in a circular neighbourhood of the physical target points.

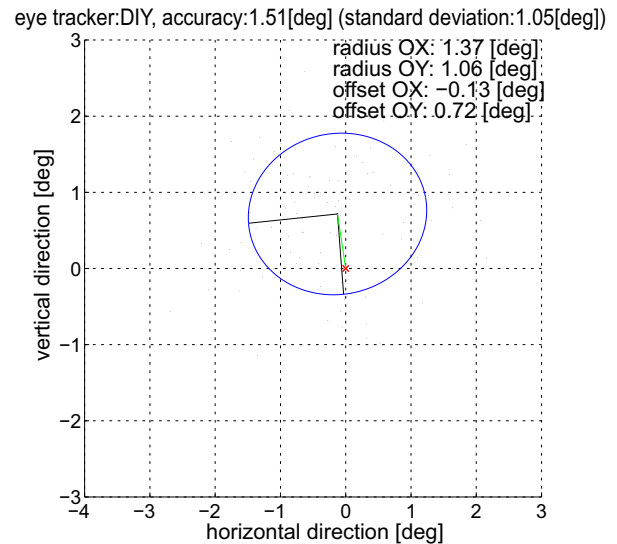


Figure 8: The average DIY eye tracker error.

We present the average eye tracker error as the covariance ellipses (see Fig. 8). The direction of the radii of ellipse corresponds to the eigenvectors of the covariance matrix and their lengths to the square roots of the eigenvalues. An eye tracker will have a good accuracy, if the distribution of the error will have the circular shape corresponding to the normal distribution and the centre of this circle will be located in (0,0) position. Additionally, the ellipse radii should be as small as possible.

As can be seen in Fig. 8, the average eye tracker error is closed to 1.51° of visual angle. The ellipse is noticeably shifted in vertical direction, which indicates the systematic error of the gaze estimation. We suspect that this error was caused by involuntary movements of the head during measurement. The DIY eye tracker is not immune to the head movements. We used the chin rest to stabilise the head but even small movements caused by e.g. swallowing could introduce some inaccuracies.

Fig. 9 shows covariance ellipse calculated for individual target points. The inaccuracies are larger for higher viewing angles. It is particularly evident for the target points located at the edges of the screen, for which the ellipses are distorted. We suspect that it was caused by occlusion of the pupil by the eye lids. Notice, that the distance from the centre of ellipse (i.e. average gaze location) is more meaningful for accuracy estimation than the radii of ellipses. For example, for (-10.1, 5.72) [deg] target point lo-

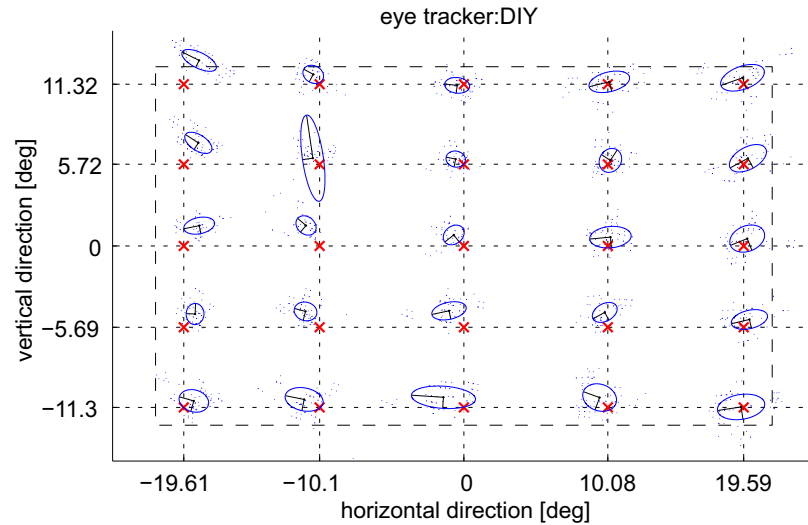


Figure 9: Covariance of the averaged locations of the gaze points recorded by the DIY eye tracker for each reference point.

cation, the distance is rather small despite the large size of the ellipse, which can be caused by some unfiltered gaze points which occurred during blinks.

5 Conclusions and future work

We have implemented a new pupil centre detection technique, which skips some time consuming operations performed by typical methods to increase the processing speed. The core of the technique is the cross spread algorithm, which detects the centre of the pupil by tracing only horizontal and vertical rays. This simple approach gives satisfactory detection accuracy for the pupil images of the centrally symmetric shape. This accuracy is comparable to ITU Gaze Tracker software results.

Our OpenCL-based implementation of the cross spread technique is executed in less than 2 milliseconds, which is equivalent to processing more than 500 frames per second. This frequency is satisfactory for accurate recording of the saccadic movements and only slightly worse than in the commercial high-end eye tracking systems.

We have integrated the cross spread method with the low-cost DIY eye tracker and tested the accuracy of this setup. The perceptual experiments have revealed satisfactory accuracy of the eye tracker equal to 1.5° per visual angle.

In future work we plan to conduct a detailed analysis of the accuracy of the cross spread algorithm. In particular, we are interested in challenging scenarios, in which the pupil is partially obscured.

References

- [1] Richard A. Abrams, David E. Meyer, and Sylvan Kornblum. Speed and accuracy of saccadic eye movements: Characteristics of impulse variability in the oculomotor system. *Journal of Experimental Psychology: Human Perception and Performance*, 15(3):529–543, 1989.
- [2] Li Dongheng, D. Winfield, and D.J. Parkhurst. Starburst: A robust algorithm for video-based eye tracking. *Proceedings of the IEEE Vision for Human-Computer Interaction Workshop*, September 2005.
- [3] Andrew T. Duchowski. *Eye Tracking Methodology: Theory and Practice (2nd edition)*. Springer, London, 2007.
- [4] R. Mantiuk, M. Kowalik, A. Nowosielski, and B. Bazyluk. Do-it-yourself eye tracker: Low-cost pupil-based eye tracker for computer graphics applications. *Lecture Notes in Computer Science (Proc. of MMM'12 Conference)*, 7131:115–125, 2012.
- [5] A. Pérez, M.L. Córdoba, A. García, R. Ménde, M.L. Muñoz, J.L. Pedraza, and F. Sánchez. A precise eye-gaze detection and tracking system. *Journal of WSCG*, 2003.
- [6] Dale Purves, George J. Augustine, David Fitzpatrick, Lawrence C. Katz, Anthony-Samuel LaMantia, James O. McNamara, and S. Mark William. *Types of Eye Movements and Their Functions. Neuroscience (2nd edition)*. Sunderland (MA): Sinauer Associates, 2001.

- [7] T. Takegami, T. Goto, and Ooyama. G. An algorithm for model-based stable pupil detection for eye tracking system. *IEICE Trans. Information and Systems* vol.J86-D-II(2), September 2003.
- [8] D. Zhu, T. Raphan, and S. T. Moore. Robust pupil center detection using a curvature algorithm. *Computer Methods and Programs in Biomedicine*, June 1999.

Automatic Synthesis of Computationally Efficient Interest Point Detectors

Ivor Uhliarik*

Supervised by: Zuzana Berger Haladová†

Faculty of Mathematics, Physics and Informatics
Comenius University
Bratislava / Slovakia

Abstract

With the arrival of the trend of integrating powerful graphics processing units into modern hand-held devices, performing complex computations is becoming feasible to the point that allows developers to deploy augmented reality-enabled smart-phone applications. This work aims to tackle the challenges of establishing an efficient pipeline of image processing tasks involved therein. We focus on the automatic synthesis of the interest point detection operator using a multiobjective genetic programming (MO-GP) framework that promotes properties suitable for detecting local features in cluttered scenes. In previous works, three properties chosen as the genetic programming (GP) search objectives have been used: stability, point dispersion, and information content. We seek to expand this approach with a fourth objective that emphasizes computational efficiency, taking parallelizability of algorithms into account. The produced operators are then validated using a set of images with appropriate content and compared with the results of existing approaches. Finally, the most promising Pareto-optimal operators are efficiently implemented in the Android RenderScript framework for use in Android mobile applications.

Keywords: augmented reality, genetic programming, interest point detection, multiobjective optimization, parallel computing

1 Introduction

Many applications of augmented reality heavily rely on the object recognition pipeline. The most frequently used method to recognize unmarked objects in unorganized, cluttered scenes is based on local image features. The process typically consists of three phases. First, salient regions in the image are detected using the *interest point (IP) detector*. Then, local features are computed for each interest point based on its local neighborhood. These features are assembled into feature vectors by *interest point*

descriptors, which try to distinctively capture the nature of objects represented by interest points. Finally, these descriptions are *matched* with precomputed descriptions of objects in a database and the nearest match is declared as the recognized object.

This approach offers resilience toward scenes where the objects are occluded or otherwise distorted. Several human-designed algorithms have emerged and proven to perform well over the last few decades, and are still an active topic in research. The drawback of this system is, however, that the task of image recognition in a general case lacks a formal definition. This results in the variance of scenarios in which different algorithms perform well. For example, some interest point detectors focus on detecting corners, while others detect edges, ridges, or blobs. Moreover, existing algorithms considerably differ in their computational complexity [14].

In recent years, there has been effort to construct algorithms used in the image recognition pipeline in an automated way using evolutionary algorithms. Olague and Trujillo in [6] have proposed a multiobjective genetic programming approach to the synthesis of interest point detectors. To counter the bias introduced in existing human-designed algorithms, this approach promotes theoretical properties the interest point detector should maximize, which are discussed in the next chapters. The outcome of this work is a set of synthesized interest point detection algorithms that exceed the recognition performance of several human-designed algorithms.

Our work seeks to build up an MO-GP framework for the synthesis of interest point detectors similar to that in [6] and extend it with a proposed novel objective that maximizes computational efficiency. The focus is laid on the feasibility of the synthesized operators to be implemented in mobile devices with competent parallel processing power, such as the modern smart-phones, whose potential is often left unexploited.

The motivation of seeking to improve the detection phase of the object recognition pipeline rests on the low-level nature and ubiquity of IP detection. The resulting algorithms may be useful for all tasks in computer vision

*ivor.uhliarik@gmail.com

†zhaladova@gmail.com

that IP detection is a part of.

Note that this is still a work in progress. The purpose is to design a framework for performing experimental research.

In Section 2, we describe the previous and ongoing work in both human-designed interest point detection algorithms and their automatic synthesis. Later in Section 3, we delve into the four qualities of detectors our MO-GP maximizes. In Section 4, we describe the genetic programming concept and explain how we optimize multiple objectives therein. Section 5 displays the results we have acquired and finally, Section 6 sums up the conclusions of our work.

2 Related Work

Human-designed algorithms for interest point detection based on local features are still the most commonly used method in the detection phase of the recognition pipeline. These can be categorized into corner and edge detectors, blob detectors, and region detectors.

An example of a corner detector is the Harris detector proposed by Harris and Stephens [2]. The approach is based on the auto-correlation matrix that describes the gradient distribution in the local neighborhood of a point. The eigenvalues of this matrix represent the principal changes in the image signal. The point for which both of the eigenvalues are large is likely to be a corner. The output of the Harris detector is shown in Figure 1. Other corner-based interest point detectors include SUSAN [9] and FAST [7].

An example of a blob detector is the scale-invariant feature transform (SIFT) by Lowe [4], which is distinguished by its extension of the image space by sub-sampling and smoothing methods to form the scale-space. This allows for scale-invariant object detection.

A comprehensive comparison of human-designed interest point detectors is discussed in [14].

The already mentioned disadvantages of human-designed algorithms are apparent: each of these solutions maximizes ad-hoc objectives. There is no universal consensus as to what defines the salience of the detected interest points. Images with smooth corners may be overlooked by corner detectors, while the points in other images may be captured more meaningfully by corner detectors rather than blob detectors.

The ad-hoc fashion of the objectives of human-designed algorithms was challenged by the work of Olague and Trujillo in 2006 [11]. A (single-objective) genetic programming approach was proposed that synthesizes interest point detectors. The approach promotes detector *stability* and *point dispersion* using a single fitness function. The output consists of several generated algorithms in form of

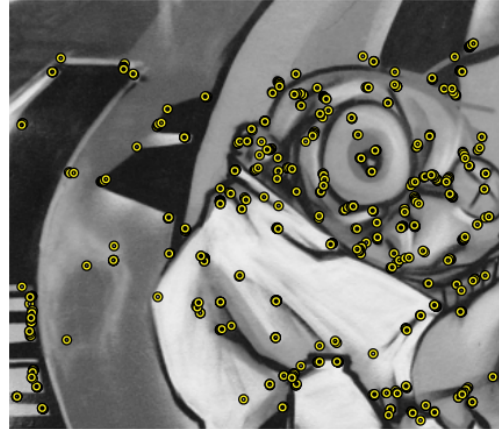


Figure 1: Example of points detected by the Harris interest point detector [14]

computational trees built up using low-level image transformations. Results of this work have been competitive to the human-designed state of the art algorithms. Later work by Trujillo and Olague in 2008 [12] showed new results that yielded performance better than many human-designed algorithms at the time using a similar setup. In 2011 [5], the same authors proposed a multi-objective GP approach, effectively splitting stability and point dispersion into two separate objectives (fitness functions). Several novel synthesized interest point detectors have been introduced. Finally, in 2012 [6], the work of Trujillo and Olague continued with the addition of a novel, third objective: *information content*.

As our proposal is based on the progressive work of Trujillo and Olague, the key concepts and algorithms will be further explained in the following sections. An example of a synthesized detector is shown in Figure 2.

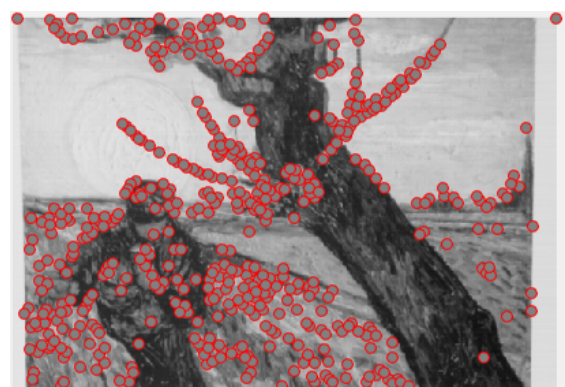


Figure 2: Example of points detected by an evolved operator synthesized by the MO-GP process using our framework

Similar work has been conducted in the synthesis of interest point *descriptors* by Liu et al. in 2013 [3]. The proposed solution uses MO-GP for the synthesis of image descriptors yielding feature vectors for detected

interest points.

3 Qualities of IP Detectors

In this section, we describe the three qualities proposed by Trujillo and Olague in [6] a good interest point detector should exhibit. Next, we propose fourth, novel objective used in our work that extends the solution.

3.1 Stability

Stability of an interest point detector is measured by its repeatability rate. In practice, this corresponds to the level of invariance of the detector toward affine transformations of the image. This objective is crucial in achieving good performance of interest operators in cluttered scenes. Given images I_1 and I_i , the set of point pairs (x_1, x_i) that are repeated in image I_i based on the image I_1 related by homography H_{1i} with maximum error of ε is

$$R_{I_i}(\varepsilon) = \{(x_1, x_i) | \text{dist}(H_{1i}x_1, x_i) < \varepsilon\}. \quad (1)$$

The overall repeatability rate is calculated as

$$r_{I_i}(\varepsilon) = \frac{|R_{I_i}(\varepsilon)|}{\min(\gamma_1, \gamma_i)}, \quad (2)$$

where $\gamma_1 = |\{x_1\}|$ and $\gamma_i = |\{x_i\}|$. $\min(\gamma_1, \gamma_i)$ in the equation represents the total number of extracted points.

The concept of the measure of repeatability is depicted in Figure 3.

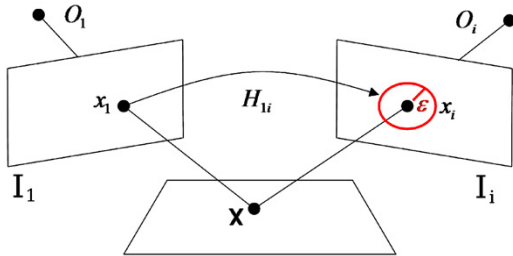


Figure 3: An illustration of the stability objective criterion [6]

3.2 Point Dispersion

The idea of uniform interest point dispersion across the coordinates of an image is simple: covering of more regions in an image is more likely to contain useful descriptions. We partition the image plane into a grid of J bins. The measure of point dispersion is defined using the entropy value of the spatial distribution of detected points X over the image plane I .

$$\mathcal{D}(I, X) = -\sum P_j * \log_2(P_j), \quad (3)$$

where P_j is approximated by the 2D histogram of the positions of interest points within bin j .

3.3 Information Content

Information content is the measure of the uniform distribution of feature vectors of the detected points. To maximize information content, therefore to avoid the loss of discriminatory power of descriptors constructed for the detected points, we have to penalize correspondences between the positional point dispersion and the descriptor space. We can achieve this by implementing the same principle as in point dispersion, but in the space of descriptors, which is illustrated in figure 4. We partition the descriptor space Γ into partitions Υ_j and approximate the probability of the occurrence of a descriptor within Υ_j by a histogram of the descriptors $\gamma \in \Upsilon_j$ as q_j . The measure of information content can then be formulated as

$$\mathcal{I}(\Gamma) = -\sum q_j * \log_2(q_j). \quad (4)$$

The choice of the descriptor algorithm used in this step is crucial for a positive effect of the objective. In [13], the SIFT descriptor has been used, which has led toward counter-intuitive MO-GP results. The problem lies in the manner in which SIFT builds the feature vector. SIFT constructs histograms of gradient orientations for a region around the interest point. This is completely appropriate when using the SIFT algorithm for interest point detection as well. In our case, though, the descriptor component is used separately. It may happen that neighboring points within regions containing curves or circles end up being drastically different in their feature vectors. For this reason, the Hölder descriptor described next is used.

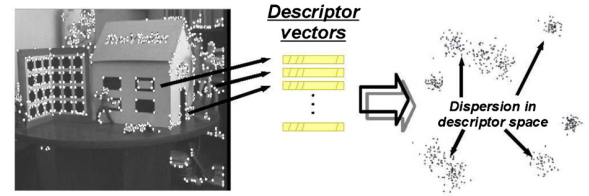


Figure 4: The effect of the objective of point dispersion on the interest point detector in a sample image [6]

3.3.1 Hölder descriptor

The Hölder descriptor is based on measuring the regularity of the region of an interest point. It follows the idea that most information is contained within irregular (singular) regions. The regularity of regions can be described by the pointwise Hölder exponent. In an image (2D signal), the exponent is defined as follows.

Pointwise Hölder exponent definition for 2D signal f
Let $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, $s \in \mathbb{R}^{++} \setminus \mathbb{N}$ and $x_0 \in \mathbb{R}^2$. Then $f \in C^s(x_0)$ if and only if $\exists \eta \in \mathbb{R}^{++}$, and a polynomial P of degree $< s$ and a constant c such that

$$\forall x \in B(x_0, \eta), |f(x) - P(x - x_0)| \leq c|x - x_0|^s, \quad (5)$$

where $B(x_0, \eta)$ is the neighborhood of x_0 with radius η . Now, the pointwise Hölder exponent of f at x_0 is defined as

$$\alpha_p(x_0) = \sup_s \{f \in C^s(x_0)\}. \quad (6)$$

The concept is better grasped when considering 1D signal as shown in figure 5. $\alpha_p(x_0)$ is the bound on the Hölder envelope—the amount by which a signal varies. Values close to zero indicate a wildly varying signal, while values close to one represent a smooth signal or a regular region. Estimators of the Hölder exponent based on genetic programming are proposed in [10]. In our work, we use the HGP-2 estimator. The application of the Hölder exponent to an image is depicted in Figure 6.

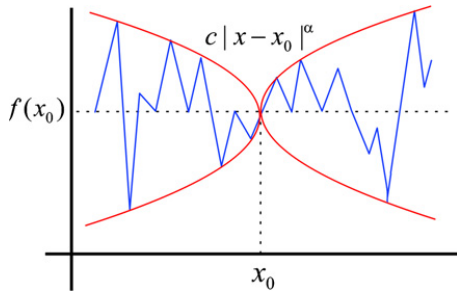


Figure 5: Visualization of the plot of the Hölder envelope of 1D signal f at point x_0 [10]

The Hölder descriptor is built by sampling the exponent at the position of each detected interest point and at equidistant positions lying on four circles of different radii around the interest point, with 32 samples per each circle. This yields a feature vector of 129 real numbers. While it is computationally costly to process such real-valued vectors in the matching phase, this problem does not concern our aim, since we only use this descriptor for the evaluation of synthesized interest point detectors, which is performed offline. The structure of the descriptor is shown in Figure 7.



Figure 6: The Hölder exponent applied to an image. The histogram of the result has been equalized for better visualization. One detected interest point is highlighted whose detail is shown in Figure 7.

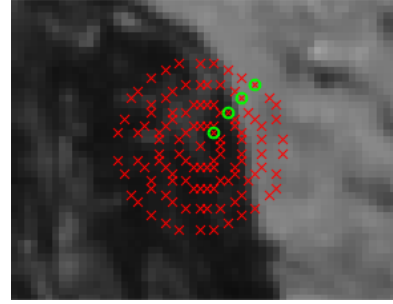


Figure 7: The structure of the Hölder descriptor applied to an image. The descriptor is formed by sampling the Hölder exponent in the image region centered at an interest point, and at 32 points per circle of 4 different radii around the interest point. The dominant gradient orientation is highlighted.

3.4 Computational Complexity

Our proposal stands on the definition of the objective promoting computational efficiency of the synthesized operators. As we focus on the implementation of the resulting operators on parallel processors¹, we should take the parallelizability of the algorithms into account.

The complexity theory defines complexity classes for problems considering their parallel nature. In particular, the class *NC* (Nick's Class) is defined as the set of problems decidable in *parallel* (polylogarithmic) time $(\log n)^{O(1)}$ on a polynomial number of processors $n^{O(1)}$ [1]. The problem with the approach using the complexity theory as a measure of fitness is that the definition is too rough for the distinction of the low-level operations used in this work (as explained in Section 4 and shown in equations 8 and 9). All of these operations are either performed independently on each point of an image, or on their local neighborhood. Therefore, all of these operations are trivially parallelizable.

The approach we propose is to empirically measure the time it takes for the synthesized algorithms to execute. This can be done in two ways.

The first involves executing the complete algorithm on a set of training images and measuring the total time of computation of the tree². The disadvantage is that the measuring capabilities are dependent on and limited to the host machine performing the GP search, and hence all synthesized algorithms are biased toward it.

The second way which is used in our work is based on taking apriori measurements of each atomic operation we use in the GP search. We are not constrained to the host machine, as these atomic operations may be imple-

¹Using the CPU, GPU, and DSP units present in a device in a heterogeneous manner.

²As generated by the MO-GP algorithm explained in Section 4

mented on the target platform containing multiple processing units, or within an emulated customizable environment. The measurements are also performed on a large set of diverse images.

Therefore, for all $op \in F \cup T$, where F and T are the sets of functions and terminals used in the GP (defined in Section 4) let $C_{op} \in \mathbb{R}$ be the measured average time it takes for op to execute. We define the computational complexity cost to be the sum of all atomic operations present in a computational tree A of the generated algorithm, as formulated in Equation 7. Note that this is a measure we are trying to minimize.

$$\mathcal{C}(A) = \sum_{op \in A} C_{op}. \quad (7)$$

4 Genetic Programming

Genetic programming is a branch of evolutionary algorithms performing symbolic regression. It is a biologically inspired optimization method based on the iterative stochastic generation of computer programs to perform a given task. With each iteration, a *population* of computer programs (computational trees) is generated or modified in a manner that aims to produce better results by preferring individuals with better survivability or *fitness* to solve the problem at hand.

This approach is similar to the genetic algorithms. Here, however, each individual consists of a computational tree with a dynamic, flexible structure. The computational trees are composed of nodes which are either *functions* (internal nodes) or *terminals* (leaves). These form the search space of the GP algorithm. The terminals act as input data that traverse upward across the tree, being transformed by every function they pass through. When the data reach the root node, the computation stops, yielding a result. Each individual in the population in this work is an image interest operator constructed by this form of symbolic regression.

In our work, we use the sets of functions F and terminals T as shown in equations 8 and 9.

$$F = \{+, |+, |-, |-, |I_{out}|, \times, \div, I_{out}^2, \sqrt{I_{out}}, \log_2(I_{out}), k * I_{out}, \frac{\partial}{\partial x} G_D, G_{\sigma=1}, G_{\sigma=2}\}, \quad (8)$$

$$T = \{I, L_x, L_{xx}, L_{xy}, L_{yy}, L_y\}. \quad (9)$$

Here, I_{out} is either one of the terminals in T , or the result of any function in F . $k = 0.05$ is a constant, G_D is the application of the Gaussian smoothing filter along direction D , and L_u is the Gaussian image derivative along direction u .

The general simplified pipeline of the computation performed by the genetic programming search is described in the following pseudo-code:

```

01 | P := generate_population()
02 | while stop condition is not met:
03 |     for individual I in P:
04 |         Fit[I] := fitness(I)
05 |     S := selection of individuals
           from P with the highest
           fitness
06 |     crossover(S)
07 |     mutation(S)
08 |     E := selection of individuals
           from P with the lowest
           fitness
09 |     P := P - E

```

When the GP search process is complete, the individuals of the population in the last iteration are considered to be good candidates of interest image operators.

An important remark is that the results of the GP search in our work are *interest point operators*. An interest point operator may be defined as a function $K(x) : \mathbb{R}^+ \rightarrow \mathbb{R}$. We may obtain the *interest image* by applying K to a regular image. Then, we say the point x is an interest point if the conditions in Equation 10 hold.

$$K(x) \geq \max\{K(x_W) | \forall x_W \in W, x_W \neq x\} \wedge K(x) > h \quad (10)$$

Here, W is a square neighborhood around point x . These two conditions represent non-maxima suppression and thresholding, respectively. In this work, we do not use a fixed value of h , but rather we choose the 500 points with the highest response to the interest operator.

4.1 Multiobjective Approach

This work aims to design and implement a framework for obtaining results that are optimal with respect to multiple optimization functions, or objectives. The MO-GP system avoids the need of manual tuning of parameters of a single fitness function to achieve results. Instead, it comes with the flexibility of altering or inserting new objectives and handles the trade-off optimization in a consolidated way. Moreover, with a single run of the GP search algorithm, we are able to obtain several non-dominant (near-optimal) results, which saves us a lot of time, considering the computational complexity of the overall MO-GP search.

The principle of the multiobjective search is intertwined with Pareto's economic theory. Given k objectives, the multiobjective space in which trade-offs and domination relations are considered, is k -dimensional and individual samples are k -dimensional vectors. In this space, we focus on finding solutions that are Pareto-optimal, i.e. are not dominated by any other vector. Considering a maximization problem, an objective vector f^i dominates objective vector f^j if no component of f^i is smaller than its counterpart in f^j and at least one component is larger. This can be seen in Figure 8 which optimizes objectives f_1 and

f_2 : the white samples are non-dominated, as there is no other sample in a dominant relation with them. On the other hand, the black samples are all dominated by at least one of the white samples. The set of non-dominated (near-optimal) solutions is called the *Pareto-front*.

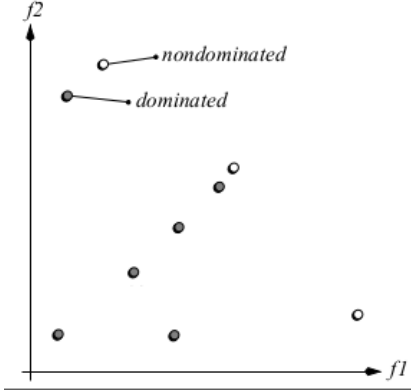


Figure 8: An example of a maximization SPEA2 search space [15]. The objective space specific to this work is depicted later in Figure 9. With all four objectives employed, this space is four-dimensional. The Pareto front represents the set of near-optimal IP detection operators that are efficient in terms of both object matching and computational complexity.

Approximation algorithms may have to deal with two problems: finding the true (optimal) Pareto-front and sampling the Pareto-front with uniform distribution across the objective space. Several multiobjective evolutionary algorithms (MOEAs) exist that solve these problems. In our work, we use Strength Pareto Evolutionary Algorithm 2 (SPEA2) [15].

5 Results

With an MO-GP framework deployed, we are able to conduct the experiments of automatic interest point operator synthesis, optimizing multiple objectives at the same time. In comparison with [6], we extend the objectives being optimized by our novel objective promoting less computationally expensive algorithms, as defined in Section 3. The solution proposed in [6] serves as a basis of comparison.

5.1 Environment Setup

To be able to objectively compare our results with the work in [6], we use the same environment and similar parameters of the MO-GP search process. The MO-GP parameters are presented in Table 1. There are only two differences. First, we set the crossover and mutation probabilities to 50%. Second, we do not limit the tree depth, as our novel objective already prefers results with lower computational cost, which is partially dependent on the tree size.

Parameter	Value
Population size	200
Generations	50
Initialization type	Ramped half-and-half
Crossover probability	0.5
Mutation probability	0.5
Mating selection	Binary tournament
SPEA2 archive size	100
SPEA2 selection size	100

Table 1: Table of parameters of the MO-GP search environment

The four fitness functions are formed of the quality measures defined in Section 3, where the first three are inverted so as to match the minimization goal. Also, the outputs of objectives 2 through 4 are empirically proportionally transformed and scaled to maintain a similar range of all four fitness values.

We use the GPLAB MATLAB toolbox [8] to perform the GP search, and the SPEA2 implementation available at the PISA website³. The training image dataset consisting of rotated Van Gogh images has been obtained from the Learning and Recognition in Vision team of Inria⁴. The repeatability rate MATLAB script from the Visual Geometry Group at Oxford University⁵ has been used.

5.1.1 Computational costs

For our computational cost objective, we have empirically evaluated the costs of atomic operations as shown in Table 2. These costs have been measured by averaging multiple runs of the operations over a random subset of 500 images of size 256x256 from the SUN scene category database⁶.

5.2 Comparison of Results

It has been expected that by extending the results in [6] with our novel objective, we yield interest operators with similar properties. The assumption is that computational complexity conflicts with other objectives in the Pareto front. This means that the more computationally efficient the operators are, the less compliant they are with the other objectives. Figure 9 shows the Pareto front of two objectives: point dispersion and stability, which gives an overview of the idea. Dispersion and stability are in conflict, which is a desired situation, as we obtain several near-optimal results with different trade-offs, all of which are useful for experimental work.

The assumption of obtaining operators with similar properties to those in [6] is confirmed as shown in figures

³<http://www.tik.ee.ethz.ch/sop/pisa/>

⁴<http://lear.inrialpes.fr/people/mikolajczyk/>

⁵<http://www.robots.ox.ac.uk/~vgg/research/>

⁶<http://sun.cs.princeton.edu/>

Operation	Time in milliseconds
f_abs	0.3237
f_const_times	0.3203
f_div	0.1594
f_gauss_1	1.6819
f_gauss_2	1.3688
f_gauss_x	1.5951
f_gauss_y	1.5466
f_log2	1.1669
f_minus	0.1358
f_minus_abs	0.1362
f_plus	0.1340
f_plus_abs	0.1385
f_square	0.3301
f_square_root	0.4282
f_times	0.1403
t_gauss_x	1.2296
t_gauss_xx	1.3685
t_gauss_xy	1.3422
t_gauss_y	1.1930
t_gauss_yy	1.3373

Table 2: Table of the evaluated computational cost of atomic operations

10 and 11. The synthesized operator in Equation 11 exposes high point dispersion fitness.

$$\frac{\partial}{\partial x} G_x(\log_2(G_{\sigma=2}(k \cdot L_y))) \quad (11)$$

This is similar to the (c) operator evolved in [6], as it also achieves high point dispersion⁷ as depicted in Figure 9. This operator uses different tree nodes, but is of similar depth and is also comprised of Gaussian filter applications as shown in Equation 12:

$$G_{\sigma=2} * \left(\frac{L_y}{L_{yy}}\right) \quad (12)$$

6 Conclusion

The aim of this work has been to implement an MO-GP framework for the synthesis of interest point detectors. The framework has successfully been implemented in a way that allows for feasible modification and insertion of objectives.

We have extended the work of Olague and Trujillo [6] by designing and implementing a fourth objective optimizing computational efficiency. This objective has been designed in a way that focuses on client runtime (mobile devices, devices with parallel computation power) rather than the host machine, while at the same time offers easy means of redefinition of the measurement of computational cost if desired.

⁷Here, the fitness is minimized, so high point dispersion measurement represents a low value in the graph

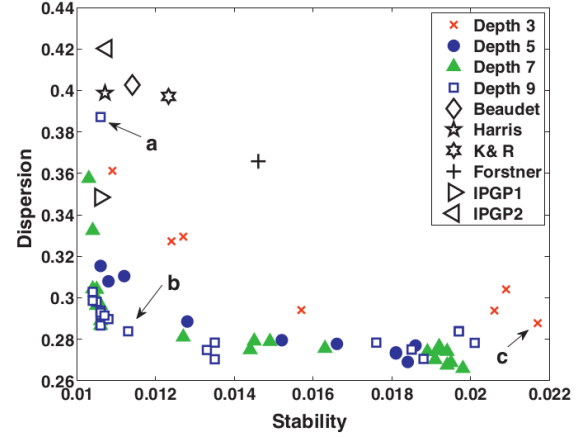


Figure 9: The Pareto front of stability and point dispersion in the work of [6]. The graph includes human-designed algorithms (Beaudet, Harris, K & R, Forstner) and operators resulting from SO-GP search presented in [11] (IPGP1, IPGP2).

In comparison, our results show high degree of similarity to those of the original work. Our intention was not to improve the results in the originally proposed three objectives, but rather to experiment by plugging the novel objective into our framework.

As mentioned in the introduction, our approach is still a work in progress. We have designed and developed the framework serving for conducting experiments. The parameters of the GP search are not definitive and heavily impact the results. Moreover, each run of the GP search algorithm takes roughly 24 hours of computation. For these reasons, we will be publishing noteworthy results of our experiments at a dedicated website⁸.

6.1 Future work

The resulting operators of this work are yet to be implemented efficiently in the heterogeneous parallel computing platform Android RenderScript.

Also, to improve the recognition in cluttered scenes, a different training image set biased toward this phenomenon may be experimented with.

Finally, we recognize that this work may be extended to generate interest point detectors usable in object recognition in 3D scenes with very little work.

References

- [1] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-completeness Theory*. Oxford University Press, Inc., New York, NY, USA, 1995.

⁸<http://davinci.fmph.uniba.sk/~uhliarik4/mogp/detector>

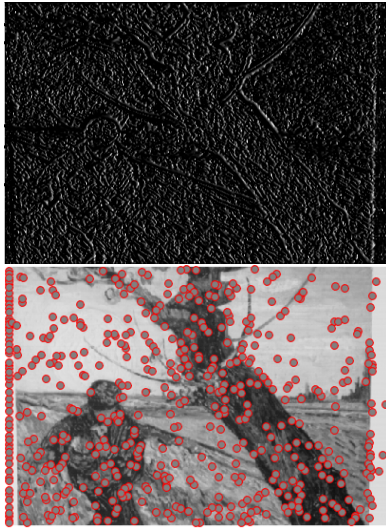


Figure 10: Interest image with detected points as the result of the synthesized operator in Equation 11, exhibiting high point dispersion measurement



(c)

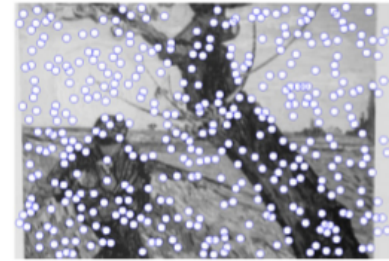


Figure 11: Interest image and points detected by the (c) operator synthesized by the MO-GP process in [6]

- [2] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, 1988.
- [3] Li Liu, Ling Shao, and Xuelong Li. Building holistic descriptors for scene recognition: A multi-objective genetic programming approach. In *Proceedings of the 21st ACM International Conference on Multimedia*, MM '13, pages 997–1006, New York, NY, USA, 2013. ACM.
- [4] David G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2, ICCV '99*, pages 1150–, Washington, DC, USA, 1999. IEEE Computer Society.
- [5] Gustavo Olague and Leonardo Trujillo. Evolutionary-computer-assisted design of image operators that detect interest points using genetic programming. *Image Vision Comput.*, 29(7):484–498, June 2011.
- [6] Gustavo Olague and Leonardo Trujillo. Interest point detection through multiobjective genetic programming. *Appl. Soft Comput.*, 12(8):2566–2582, August 2012.
- [7] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *Computer Vision-ECCV 2006*, pages 430–443. Springer, 2006.
- [8] Sara Silva and Jonas Almeida. Gplab-a genetic programming toolbox for matlab. *Proceedings of the Nordic MATLAB conference*, pages 273–278, 2003.
- [9] Stephen M Smith and J Michael Brady. Susana new approach to low level image processing. *International journal of computer vision*, 23(1):45–78, 1997.
- [10] Leonardo Trujillo, Pierrick Legrand, Gustavo Olague, and Jacques Lévy-Vehel. Evolving Estimators of the Pointwise Holder Exponent with Genetic Programming. *Information Sciences*, 209:61–79, 2012. Submitted.
- [11] Leonardo Trujillo and Gustavo Olague. Synthesis of interest point detectors through genetic programming. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, GECCO '06*, pages 887–894, New York, NY, USA, 2006. ACM.
- [12] Leonardo Trujillo and Gustavo Olague. Automated design of image operators that detect interest points. *Evol. Comput.*, 16(4):483–507, December 2008.
- [13] Leonardo Trujillo, Gustavo Olague, Evelyne Lutton, and Francisco Fernández de Vega. Multiobjective design of operators that detect points of interest in images. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, GECCO '08*, pages 1299–1306, New York, NY, USA, 2008. ACM.
- [14] Tinne Tuytelaars and Krystian Mikolajczyk. Local invariant feature detectors: A survey. *Found. Trends. Comput. Graph. Vis.*, 3(3):177–280, July 2008.
- [15] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. Technical report, 2001.

Automatic Detection of Shadow Acne and Peter Panning Artefacts in Computer Games

Rafał Piórkowski*

Supervised by: Radosław Mantiuk[†]

West Pomeranian University of Technology, Szczecin
Poland

Abstract

Contemporary game engines offer an outstanding graphics quality but they are not free from typical graphics artefacts. Essential deteriorations are the shadow acne and peter panning artefacts related to deficiency of the shadow mapping technique. In this work we assess whether the objective image quality metrics (IQMs) are suitable for automatic detection and evaluation of these artefacts. We conduct subjective experiments in which people manually mark the visible local artefacts. Then, the detection maps averaged over a number of observers are compared with results generated by IQMs. We evaluate effectiveness of the mathematically-based objective metric - MSE, and advanced IQMs: S-CIELAB, SSIM, MSSIM, and HDR-VDP-2. The achieved results reveal that MSSIM and SSIM metrics outperforms other techniques and are the most suitable for automatic detection of the shadow acne and peter panning.

Keywords: image quality metrics, game engine artefacts, shadow acne, peter panning, perceptual experiments, image quality

1 Introduction

Graphics artefacts are anomalies found in rendered images. They can significantly degrade an image reception and reduce the overall quality of graphics. Interestingly, contemporary advanced game engines are not free from presence of the visually confusing artefacts. In this work we evaluate two types of such deteriorations: *shadow acne* and *peter panning*. Shadow acne (see Fig. 1) is caused by limited depth resolution of the depth maps used in the shadow maps technique [1, Sect.Shadow map]. This artefact can be reduced applying the bias shift to the depth computation. However, too excessive displacement can cause the discontinuity of shadows, i.e. the peter panning deterioration (see Fig. 3). The latter one does not degrade the graphics quality directly but can be perceived by humans as something unnatural.

Our goal in this paper is to find out whether the objective image quality metrics (IQMs) [7] are suitable for detection of the shadow acne and peter panning artefacts. The primary application of this concept is an automatic detection of the artefacts during the game production process. Another important issue is evaluation of the perceptual importance of an artefact. If it is barely visible for human observers its correction can be neglected to save the GPU resources.

The image quality assessment revealed its usefulness in the computer graphics applications. The extensive studies were performed in the area of 3D mesh quality assessment [7]. The mesh simplification causes such artefacts as geometric quantisation noise or texture deteriorations. The first attempts to evaluate the visual fidelity of these types of artefacts were simple geometric distance metrics [12]. The advanced IQMs were also tested with the conclusion that better detection of the mesh simplification deterioration can be achieved using the model-based metrics [10]. A comprehensive review of other assessment techniques in this field has been published in [6].

Rushmeier et al. [11] studied the effectiveness of replacing geometric detail with texture maps as a method of simplification. They used a psychophysical scaling procedure to measure the perceived fidelity of simplified geometry and textures relative to the reference representation. They focused on a user study and analysis of its results rather than using the objective metrics.

In [2] a quality metric for stereoscopic images was proposed. It combines the typical 2D image quality metric (SSIM or C4) with the depth information. Another idea was presented in [9], in which the depth map is compressed based on the results of the visual masking experiment. Differences in depth, which are invisible to the human and not caused the visible artefact in the stereo images, are masked out to reduce the size of the depth map.

We focus on the static artefacts that are visible in a separate frame of the game animation. Even more prominent are the artefacts occurring in the temporal domain, that cause the flickering. Analysis of this type of deteriorations requires different quality metrics and a separate experimental methodology (see examples in [13, 6]). We address this issue to future work.

In this paper we describe conducted subjective experi-

*rpiorkowski@wi.zut.edu.pl

[†]rmantiuk@wi.zut.edu.pl

ment in which observer manually marks the visible local artefacts in snapshots from the games. It is done in the presence of the reference image without artefacts (ground truth). Hence, we perform the full-reference experiment compatible with the full-reference IQMs. This methodology follows the technique introduced in [5] and [4]. However in the mentioned papers, Čadík et al. [5] evaluate artefacts caused by limitation of the global illumination rendering techniques. The shadow acne and peter panning have a different characteristic and are common for the real-time rendering systems. We evaluate if they are identified by simple arithmetic difference (MSE), colour difference metric S-CIELAB [18], texture statistics SSIM [17], MSSIM [15], and metric based on perceptual models HDR-VDP-2 [8].

The achieved results reveal that MSSIM and SSIM metrics outperform other techniques and are the most suitable for automatic detection of the shadow acne and peter panning artefacts.

The paper is organised in the following way. In Sect. 2 the shadow acne and peter panning are outlined. Sect. 3 presents details on the conducted perceptual experiments. We compare the detection maps marked by human observers with the maps generated by IQMs in Sect. 4. In this section we also briefly describe the advantages of individual objective metrics (Sect. 4.2). The paper ends with conclusions and providing directions for further work in Sect. 5.

2 Shadow acne and peter panning

In this section two prominent graphics artefacts are presented: shadow acne and peter panning. We discuss the reasons for their occurrence in the graphics engines and how to prevent them from occurring.

Shadow acne

Shadow acne also called *erroneous self-shadowing*, may occur when shadow depth map algorithm [1, Sect. Shadow map] is used in order to add shadows into the scene in the real-time graphics engine. This artefact manifests itself as moire patterns on surfaces (see Fig. 1). The shadow maps technique consists of two passes. In first, scene is rendered from the light source point of view. Information about the distances between light source and objects is stored as texture called *shadow map*. Those distances are called depth. The more distant is the object from the light source the brighter is texel in the shadow map. During the second pass, when the scene is rendered from the camera point of view the location of each pixel is compared to the corresponding texel in the shadow map. If a rendered point is farther away from the light source than the corresponding value in the shadow map, that point is in the shadow, otherwise it is not.

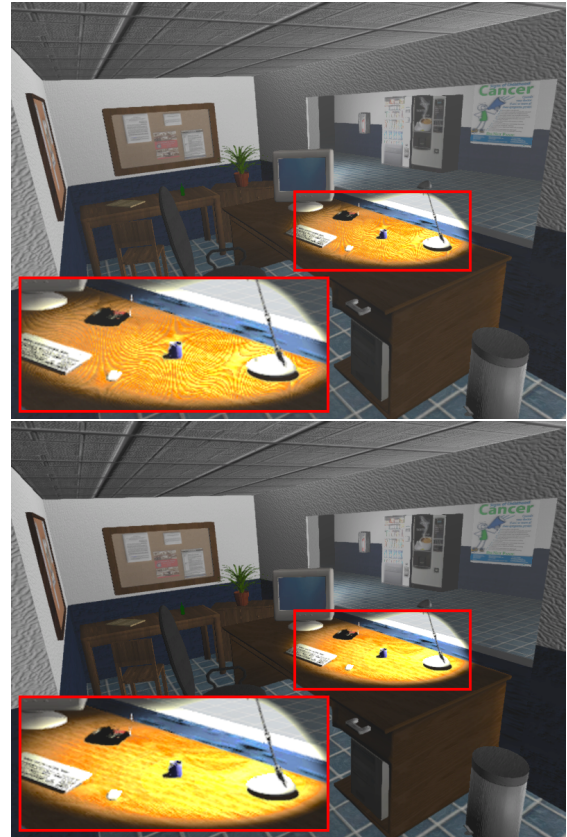


Figure 1: Shadow acne artefacts (top) and the corrected frame (bottom). The inset depicts the surface which is self-shadowed but it should not be shadowed at all.

The shadow acne artefact can be caused by two factors. The first reason is a limited computation precision of the depth maps. When both depth values compared in the second phase of the shadow map algorithm are close to each other, the depth test may fail for selected pixels. The second issue is geometrical - shadow map quantises the depth over an entire texel (see Fig. 2), while a surface is smooth. Due to this fact, a depth test can erroneously give over- and under-the-surface result for the same texel, resulting in self-shadowing.

The most common method to mitigate shadow acne artefact is adding small value - *bias* into light space when depth test is computing (see Fig. 2).

Peter panning

Peter panning is another artefact connected with shadow depth maps algorithm (see Fig. 3). This term derives from the book character - Peter Pan, who could fly and his shadow was detached from body. When this artefact occurs, shadow is detached from the object which seems to hover above surface. Peter panning appears when too large bias is used to prevent the shadow acne occurrence.

Finding proper bias value in order to simultaneously avoid shadow acne and peter panning artefacts for whole

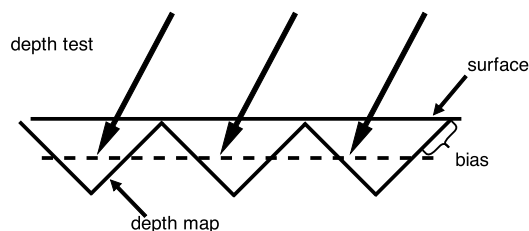


Figure 2: Depth values are stored in the depth map with limited spatial precision. Adding *bias* shift prevents erroneous depth tests.

scene and each frame could be computationally expensive and affect on performance.

3 Experimental study

The goal of the experiment was to create the reference maps that identify the artefacts seen by the people in the game screenshots.

3.1 Stimuli

Even the most prominent and popular graphics engines are not free from the rendering artefacts. We selected three contemporary graphics engines that deliver the development environment for independent developers: Unity 3d ¹, CryEngine 3 ², and Unreal Engine 4 ³. In these engines it was possible to model a scene based on external graphics objects or some examples delivered with the engine. Then we ran the game changing the rendering parameters. In particular, the shadow mapping was activated with different bias levels to test the shadow acne and peter panning deteriorations.

We modelled 20 different scenes. We used the static camera to avoid motion in the scene. Scene objects and game engine parameters were combined in a way resulted in the appearance of shadow acne or peter panning artefacts. It was done using the *bias* coefficient that was set to too low value for the shadow acne and too high for the peter panning. In our stimuli for 10 scenes we forced the shadow acne and in remaining 10 the peter panning. For each scene the reference image was generated with the correct *bias*. In real-world games it is often challenging to automatically find a correct *bias*, which can differ for various scenes and even various camera shots.

The screenshots of the scenes were captured using the FRAPS program ⁴, which saved images in 800x600 pixel resolution.

¹<http://www.unity3d.com>

²<http://www.cryengine.com>

³<http://www.unrealengine.com>

⁴<http://www.fraps.com>

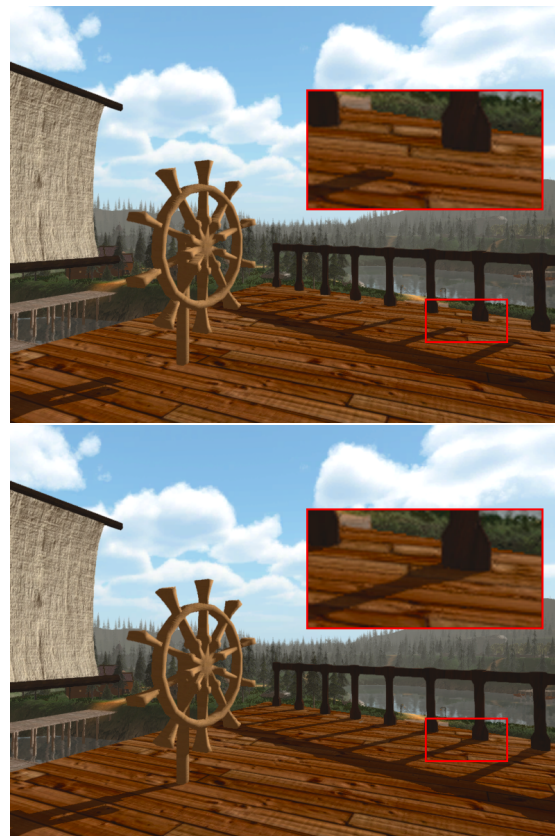


Figure 3: Peter panning artefact (top) and the reference frame (bottom). The shadow discontinuity can be seen in shadows cast by the railing posts.

3.2 Experimental procedure

We asked people to manually mark visible differences between the reference image and an image with a particular artefact. Observers used a custom brush-paint interface controlled by the computer mouse. The brush size could be reduced up to per-pixel resolution. This procedure was repeated for every scene, resulting in 20 comparisons and finally 20 binary difference maps generated per observer.

The experiment was performed in a darkened room. Images were displayed on 24" Eizo ColorEdge CG245W monitor with native resolution of 1920 x 1200 pixels. This display is equipped with the hardware colour calibration module and was calibrated before each experimental session to sRGB colour profile with the maximum luminance level increased to 110 cd/m². During the experiment, an observer was sitting in front of the display at a distance of 70 cm. This distance was not stabilized by a chin rest but we asked observers to keep it approximately constant.

3.3 Participants

We repeated the experiment for 25 volunteer observers (age between 20 and 23 years, 23 males and 2 females). They declared normal or corrected to normal vision and

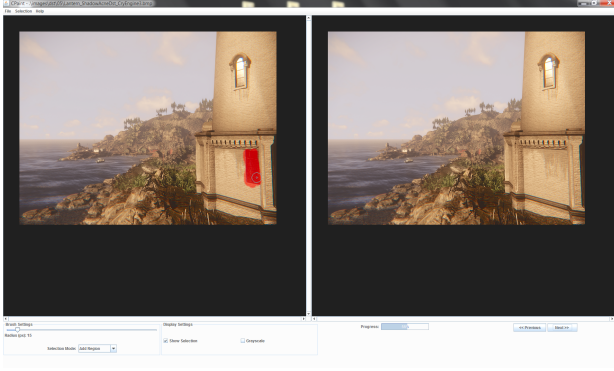


Figure 4: Screenshot from the experiment. Notice the red vertical mark in the left image made by an observer.

correct colour vision. The participants were aware that the image quality is evaluated, but they were naïve about the purpose of the experiment.

The experiment is time consuming, therefore we clustered the stimuli images into packages consisting of 10 pairs of images (tested and reference). While there were no time limitations to our study, the average subject finished marking a package in approximately 15 minutes.

4 Results

A goal of the experiment was to test which of the full-reference IQMs is the most suitable for testing the game engine artefacts. We achieved the *reference difference maps* from results of the perceptual experiment described in Sect. 3 and analysed in Sect. 4.1. These results were compared with the *test reference maps* generated by IQMs in Sect. 4.2.

Both reference and test maps are compared using the receiver-operator-characteristic (ROC) technique and their coherence was expressed as the Area Under Curve (AUC) value (see Sect. 4.3). The whole procedure is outlined in Fig. 5.

4.1 Reference difference maps

Example difference maps created by a single observer during the experiment are presented in Fig. 6. The white background represents untouched pixels while the pixels marked by observer are drawn in grey. Latter pixels depict the areas in the test image recognised as artefacts by the human observer.

Kendall analysis

The Kendall rank correlation coefficient (or Kendall's tau (τ)) is a statistic used to measure the association between two measured quantities. In our case, it assesses the inter-observer agreement, i.e. the similarity of the difference maps created by individual observers. As shown by Cadic

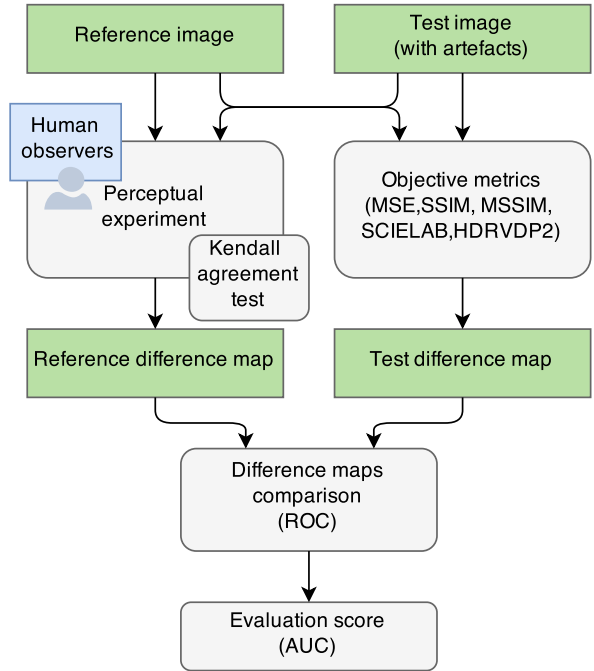


Figure 5: Evaluation procedure.

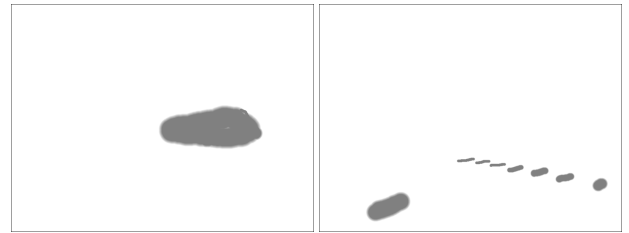


Figure 6: Example difference maps created by a single observer for images with the shadow acne (left) and piter panning (right) artefacts.

et al. [5], we used the τ value to assess whether people marked similar areas for a given pair of test and reference images. The coefficient τ ranges from $\tau = -1/(o - 1)$, which indicates no agreement between o observers, to $\tau = 1$ indicating that all observers responded the same. Examples of the coefficient maps are shown in Fig. 7.

We computed average coefficients $\bar{\tau}$ for each scene containing artefacts. However, these values tend to be skewed toward very high values because most pixels did not contain any distortion and were consistently left unmarked by all observers. Therefore, we also compute a $\bar{\tau}_{masked}$, which considers only those pixels that were marked as distorted by at least two observers. We achieved $\bar{\tau}$ equal to 0.97 and $\bar{\tau}_{masked}$ to 0.44, averaged over all scenes. These values indicate a high inter-observer agreement. For comparison, in the similar experiment described in [5] and assumed as a high consistent, the $\bar{\tau}$ and $\bar{\tau}_{masked}$ equaled to 0.78 and 0.41, respectively.

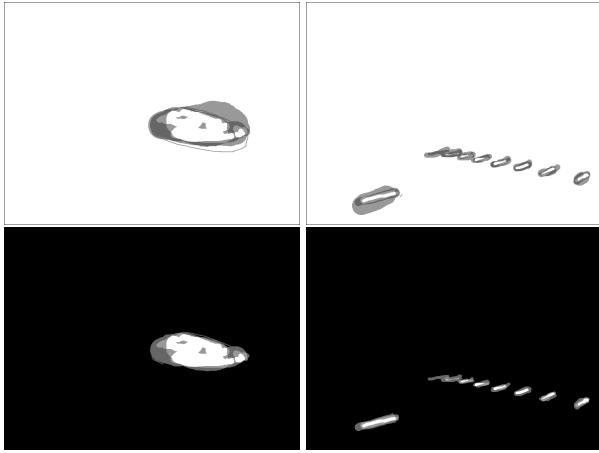


Figure 7: Examples of Kendall coefficient maps (top row) and Kendall maps after masking (bottom row). The white pixels depict good agreement between observers. The maps correspond to shadow acne example from Fig. 1 (left column), and peter panning from Fig. 3 (right column).

Averaged difference maps

We averaged the difference maps related to individual test image over all observers to achieve the reference difference maps. Then, these maps were binarised with the 0.5 threshold. In other words, the pixels marked by 50% of observers were set to 1 and remaining pixels to 0. This thresholding gives reliable result during further statistical analysis, because it eliminates strong deviations in markings. Example reference difference maps are shown in Fig. 8.

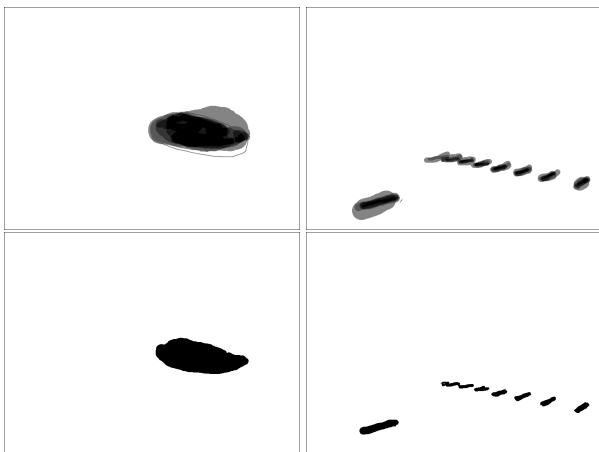


Figure 8: Example difference maps averaged over all observers before binarization (top row) and after (bottom row). Presented artefacts are shadow acne (left column) and peter panning (right column).

4.2 Objective metrics

Objective Image Quality Metrics (IQM) deliver quantitative assessment of the perceptual quality of images [14][16]. In our studies we chose four representative IQMs: S-CIELAB (Spatial-CIELAB), SSIM (Structural SIMilarity Index), MSSIM, and HDR-VDP-2 (High Dynamic Range Visual Difference Predictor) that prove their efficacy in perceptual comparison of images. Additionally, we evaluated the results of the MSE metric to give a background for comparison. The S-CIELAB metric [18] is a spatial extension of standard CIELAB colour difference. SSIM [17] and MSSIM [15] detect structural changes in the image. They are sensitive to difference in the mean intensity and contrast but the main factors are local correlations of pixel values. These dependencies carry information about the structure of the objects and reveal structural image difference between tested and reference images. HDR-VDP-2 [8] predicts the quality degradation expressed as a mean opinion score of the human observers and visibility (detection/discrimination) of the differences between tested and reference images. It takes into account the contrast sensitivity function measured for variable background luminance and spatial frequencies. The sensitivity to light is modelled separately for cones and rods resulting in correct prediction for mesopic and scotopic light conditions.

For each IQM, we generated 20 test difference maps that were compared to corresponding reference difference maps. Example maps computed using each mentioned metric are presented in Fig. 10 and Fig. 9.

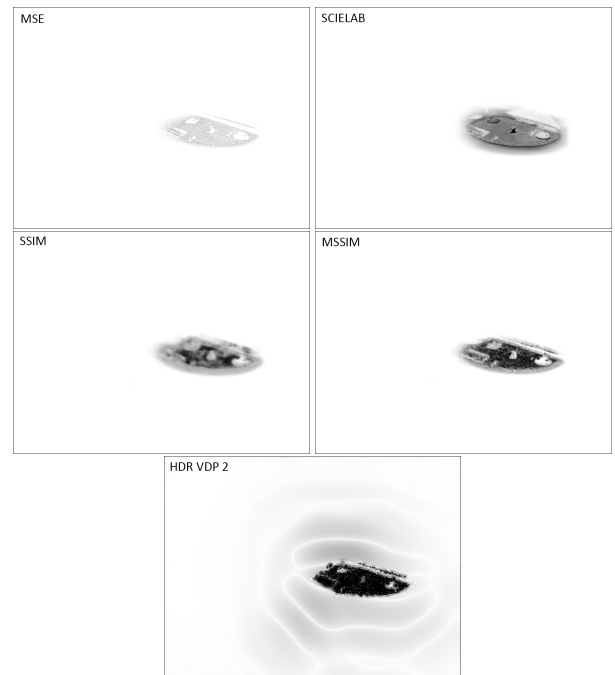


Figure 9: Difference maps automatically generated by IQMs for the shadow acne artefact.

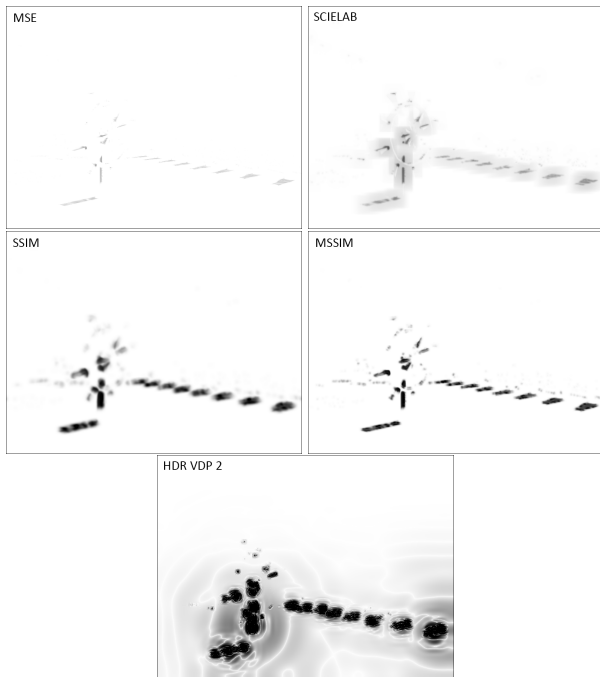


Figure 10: Difference maps automatically generated by IQMs for the peter panning artefact.

4.3 ROC analysis

The key question is whether any of the IQM performs significantly better than the others in terms of detecting the shadow acne and peter panning artefacts. In our experiment, observers binary classified pixels that contained artefacts. The performance of such classification can be analysed using the receiver-operator-characteristic (ROC) [3]. ROC captures the relation between the size of artefacts that were correctly marked by a IQM (true positives), and the regions that do not contain artefacts but were still marked (false positives). The metric that produces a larger *area under the ROC curve* (AUC) is assumed to perform better.

The ROC plots for individual metrics are presented in Fig. 11. We achieved the best results for the SSIM and MSSIM metrics (AUC=99.38 and 98.69, respectively). Interestingly, simpler metric SCIELAB works comparatively well (AUC=99.33). As it was expected MSE gave the worst results with AUC=89.34. We achieved also poor result for HDR-VDP-2 (AUC=90.26). This metric incorrectly detected artefacts covering the large areas, which is common e.g. for the peter panning. Analysis of this issue we address for further work.

5 Conclusions

In this work we asked group of observers to find local artefacts in images rendered by the real time game engines. We focused on two types of artefacts: shadow acne and peter panning, accompanying the shadow mapping tech-

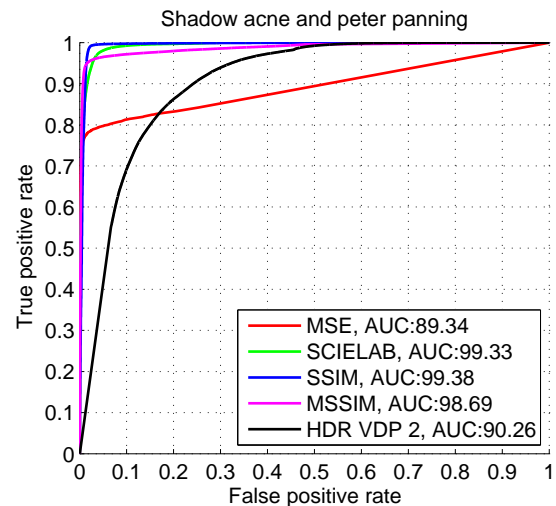


Figure 11: Results of the ROC analysis for individual IQMs.

nique. The reference difference maps derived from this perceptual experiment were compared with the difference maps generated by the most recognised objective image quality metrics. The ROC analysis revealed the best accuracy of the SSIM metric with the effectiveness of detection close to 100%.

In future work we plan to analyse other artefacts, especially aliasing and z-fighting. We are also interested in the flickering resulting from the motion on the scene. However, localised analysis of such artefacts seems to be challenging.

References

- [1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. A K Peters, Ltd., Wallesley, Massachusetts, third edition, 2008.
- [2] Benoit Alexandre, Le Callet Patrick, Campisi Patrizio, and Cousseau Romain. Quality assessment of stereoscopic images. *EURASIP journal on image and video processing*, 2008, 2009.
- [3] P. Baldi, S. Brunak, Y. Chauvin, C. A. F. Anderson, and H. Nielsen. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics*, 16(5):640–648, 2000.
- [4] Martin Čadík, Robert Herzog, Rafał Mantiuk, Radosław Mantiuk, Karol Myszkowski, and Hans-Peter Seidel. Learning to predict localized distortions in rendered images. In *Computer Graphics Forum*, volume 32, pages 401–410. Wiley Online Library, 2013.
- [5] Martin Čadík, Robert Herzog, Rafał Mantiuk, Karol Myszkowski, and Hans-Peter Seidel. New measurements reveal weaknesses of image quality metrics in

- evaluating graphics artifacts. *ACM Transactions on Graphics (TOG)*, 31(6):147, 2012.
- [6] Massimiliano Corsini, Mohamed-Chaker Larabi, Guillaume Lavoué, Oldřich Petřík, Libor Váša, and Kai Wang. Perceptual metrics for static and dynamic triangle meshes. In *Computer Graphics Forum*, volume 32, pages 101–125. Blackwell Publishing Ltd, 2013.
- [7] Guillaume Lavoué and Rafał Mantiuk. Quality assessment in computer graphics. *Visual Signal Quality Assessment*, pages 243–286, 2015.
- [8] Rafał Mantiuk, Kil Joong Kim, Allan G. Rempel, and Wolfgang Heidrich. Hdr-vdp-2: A calibrated visual metric for visibility and quality predictions in all luminance conditions. *ACM Trans. Graph.*, 30(4):40:1–40:14, July 2011.
- [9] Dawid Pajak, Robert Herzog, Radosław Mantiuk, Piotr Didyk, Elmar Eisemann, Karol Myszkowski, and Kari Pulli. Perceptual depth compression for stereo applications. *Computer Graphics Forum*, 33(2):195–204, 2014.
- [10] Bernice E Rogowitz and Holly E Rushmeier. Are image quality metrics adequate to evaluate the quality of geometric objects? In *Photonics West 2001-Electronic Imaging*, pages 340–348. International Society for Optics and Photonics, 2001.
- [11] Holly E Rushmeier, Bernice E Rogowitz, and Christine Piatko. Perceptual issues in substituting texture for geometry. In *Electronic Imaging*, pages 372–383. International Society for Optics and Photonics, 2000.
- [12] William J Schroeder, Jonathan A Zarge, and William E Lorensen. Decimation of triangle meshes. In *ACM Siggraph Computer Graphics*, volume 26, pages 65–70. ACM, 1992.
- [13] L. Vasa and V. Skala. A perception correlated comparison method for dynamic meshes. *IEEE Trans. on Visualization and Computer Graphics*, 17(2):220–230, 2011.
- [14] Zhou Wang and Alan Bovik. *Modern Image Quality Assessment*. Morgan & Claypool Publishers, 2006.
- [15] Zhou Wang, Eero P Simoncelli, and Alan C Bovik. Multiscale structural similarity for image quality assessment. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Seventh Asilomar Conference on*, volume 2, pages 1398–1402. Ieee, 2003.
- [16] H. Wu and K. Rao. *Digital Video Image Quality and Perceptual Coding*. CRC Press, 2005.
- [17] Wang Z., Bovik A.C., Sheikh H.R., and Simoncelli E.P. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions On Image Processing*, 13(4):600–612, 2004.
- [18] X. M. Zhang and B. A. Wandell. A spatial extension to cielab for digital color image reproduction. *Proceedings of the SID Symposiums*, pages 731–734, 1996.

3D Reconstruction

Adapting Hair and Face Geometry of Virtual Avatars with the Kinect Sensor

Hannes Plank

Supervised by: Stefan Hauswiesner

Graz University of Technology, Austria

Abstract

In this project, we developed a method for creating virtual head models that look like the user. The resulting application scans the user with a Microsoft Kinect sensor to obtain RGB-D images. Based on these images, a virtual face model can be adapted to resemble the users appearance.

A fast C++ implementation was created. To generate hair, several hair parameters were obtained. A method to personalize existing hair mesh templates was implemented.

1 Introduction

Virtual avatars are virtual 3D representations of humans. They can help users to feel more involved with an application and enable a number of future high-impact use cases. For example, users can see themselves in different clothes in virtual dressing rooms. Computer games could be customized to allow the user to be the main character of a game, or interact with other people and their personal avatars. High-end teleconferencing systems would allow participants to collaborate naturally in a 3D virtual meeting room.

We developed an implementation capable of creating virtual avatars using the popular Microsoft Kinect sensor. The implementation acquires several RGB-D frames of the user. Using a face tracking algorithm, it tracks the users head and fits a face tracking mesh for every frame. The gathered RGB-D frames are converted to 3D pointclouds. Using the head tracking information, the pointclouds are placed over a generic morphable 3D face. The avatar is morphed with an optimization algorithm, minimizing the distance between the avatar and the pointcloud.

To ensure compatibility with common work-flows, we use avatar data models that can be generated using DAZ 3D Studio. These avatars feature parametric head models which are customizable.

To generate realistic hair, an algorithm detects various hair parameters. Several hair template meshes for different hair lengths were created. By evaluating the hair parameters, the best template is chosen automatically. The template is morphed to adapt to the persons hairstyle.

The hair and face meshes are projectively textured by

using the best captured RGB image.

To demonstrate the algorithm, a fast C++ implementation, using the Kinect SDK and OpenCV was developed.

2 Related Work

The Microsoft Kinect was the first affordable depth camera, available on the consumer market. Since the Microsoft Windows driver appeared, a lot of Kinect and RGB-D camera related research was accomplished.

There is an approach to scan a whole person with the Kinect [Sum+13]. This method however needs 15 minutes to generate a watertight mesh, with an algorithm similar to Kinect Fusion [Iza+11]. Our project is focused on making the avatar personalization process as intuitive and simple (from the users perspective) as possible.

Zollhofer et al. [Zol+14] present a very similar approach to our project. They built an interactive system which reconstructs facial data in real time, while giving the user feedback. Instead of morphing a pre-designed face template mesh, like in this project, this implementation combines 200 different heads into a statistical shape model. With depth fitting and regularization, they estimate the parameters of the head. We apply morphing for our implementation, which replaces the task of creating a statistical model.

If there is no depth information available, Jiang et al. [Jia+05] show that it is also possible to reconstruct faces by using 3D face shape databases. In their approach, the 2D face image is first aligned with a generic 3D face geometry. Since all face geometry is compressed by PCA, the key features of the 2D face are used to compute the 3D shape coefficients of the Eigen vectors. The face shape is reconstructed by using these coefficients.

Cao et al. [Cao+13] show, that animating abstract avatars can be performed just by using simple VGA cameras. It is possible to transfer facial expressions from a person in real time to any mesh. The paper demonstrates how far interaction with personalized avatars can go, and how much potential there is. As first step, the implementation requests the user to make several extreme facial expressions. The abstract avatars have geometrical models of each these extreme facial expressions. In the interactive initialization part, the algorithm registers the persons

extreme facial expressions. During animation, it interpolates the previously saved extreme expressions. This is also possible with the avatars created by our project, however additional face meshes for the extreme facial expressions would be required.

Chai et al. [Cha+13] show a very promising way to generate 3D volumes of human hair just by using an image sequence. The gathered hair model is based on strands. The model also enables physically plausible animation of hair. As mentioned in section 5, the image quality of the Kinect color camera was not sufficient to consider this hair generation method.

Blanz et al. [BV03] also use 2D images and a 3D scan database to obtain face geometry. Their implementation takes several 2D images of the same face to fit data to a morphable 3D face model. Their morphable 3D face model was created from a database of 3D scans. The goal of this 3D scanning methods is face recognition rather than realistic avatar generation.

There is also a way to reconstruct faces from a single 2D image with a generic face mesh [Mag+13]. It exploits the global similarity of faces and combines shading information with generic shape information. With a Kinect depth camera available, there is not much potential to use this technique additionally. However some geometry adaptations, mentioned in section 5.3, are only influenced 2D texture evaluation.

3 The Procedure

The virtual avatars are created by using RGB-D data to morph a generic face mesh. At first, multiple RGB-D frames are captured with the Kinect sensor. They are converted to pointclouds and are aligned with a morphable face mesh.

After morphing the head, some processing is done to improve the avatar and add hair geometry. Additional processing steps include the projective texture mapping, and positioning the eyes.

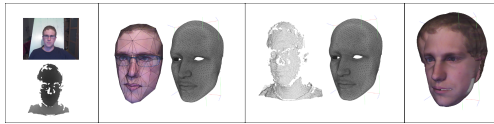


Figure 1: The acquired data, the facetracking mesh aligned with the morphable generic head mesh, the pointclouds aligned with the head mesh, morphed result with eye and hairmesh.

3.1 Data gathering

The data gathering software captures the RGB-D frames and tracks the rotation and position with a head tracking algorithm [Mic13]. Several frames are captured by the Kinect RGB-D camera. The user presses a key to capture a

frame in a ten second interval. The optimal distance from the sensor is one meter. Best results are obtained with a bright diffuse illumination and a neutral background.

The following data is captured per frame:

- A color image.
- The depth information. Using the known camera parameters, a depth image is converted to a 3D pointcloud. Due to the face tracking, it is possible to align the pointclouds.
- A facetracking mesh. The Kinect facetracker fits a Candide 3 [Ahl01] mesh to the face of the user. The resulting low polygon facetracking mesh has the same orientation and scale of the pointclouds.
- The rotation and translation of the head, calculated by the Kinect facetracking algorithm.
- A projection matrix for projective texturing.

3.2 Face mesh production

The implementation goes through the following processing steps as illustrated in Figure 1:

1. The Candide 3 headtracking meshes are in the same coordinate system as their corresponding pointclouds. Each pointcloud has one headtracking mesh, so the pose of the users head is saved in the pose of the headtracking mesh. The orientation and position of the facetracking meshes is used to calculate a transformation for each point cloud. The transformed pointclouds are aligned in the same coordinate system. The morphable face template mesh is scaled and placed close to the pointclouds..

$$2. \quad \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} * f \quad (1)$$

The morphing equation (1) transforms a vertex v in direction d by a factor f . The external morphing data from [Pro14] contains groups of vertices with the same factor f_n . Each vertex is assigned a direction vector v_n . The direction vector's magnitude defines how much influence a change in factor f_n has.

To be able to morph the head template, all morphing equations are compiled into equation system (2).

$$\begin{pmatrix} p_{1x} & 0 & 0 & \vdots \\ 0 & p_{1y} & 0 & \vdots \\ 0 & 0 & p_{1z} & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} = \begin{pmatrix} v_{1x} & 0 & 0 & \vdots \\ 0 & v_{1y} & 0 & \vdots \\ 0 & 0 & v_{1z} & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} + \begin{pmatrix} f_1 & 0 & 0 & \vdots \\ 0 & f_1 & 0 & \vdots \\ 0 & 0 & f_1 & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} \quad (2)$$

The morphing factors are retrieved by solving the equation system. This can be seen as a minimization of the distances between the vertices of template

mesh v and their nearest neighbours p in the pointclouds. When the morphing factors are known, the morphing equation 1 can be applied to each vertex v . This enables to create a personalized, watertight DAZ 3D face mesh. This format can be incorporated into existing workflows.

3. The mesh is projectively textured with a color image and saved in Wavefront obj mesh file format.

4 Improvements

The avatars generated by the introduced algorithm resemble the scanned person, however a better result is received by some improvements.

Precise alignment of the pointcloud data and the morphing template is crucial for good results. The mesh produced by the Kinect face tracker, is in the same coordinate system as the pointcloud data. This means the necessary pointcloud transformation can be calculated by aligning the face tracking mesh with the morphing template. At first the generic unfitted Candide 3 [Ahl01] face tracking mesh was used. To take individual face proportions into account, a fitted Candide 3 mesh as seen in Figure 2, was used instead. This leads to better pointcloud alignment and improves fitting the textures.

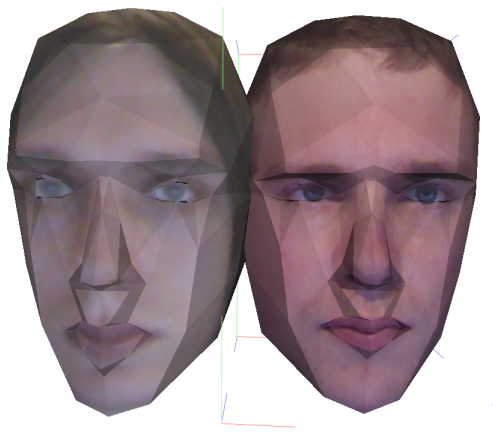


Figure 2: Fitted Candide 3 facetracking meshes, obtained by the Kinect SDK facetracker.

Aligning the facetracking mesh with the morphable facemesh is not trivial. Various alignment methods were evaluated. Scaling along all dimensions did not produce realistic looking faces, since the persons face proportions got distorted. When the aligning aims to minimize the distance to the facial features, the rest of the head is distorted tremendously.

Simply aligning the face tracking mesh by its center and scaling by a calculated factor in every dimension as seen in Figure 3, produced the best results. The result of the alignment is shown in Figure 4.

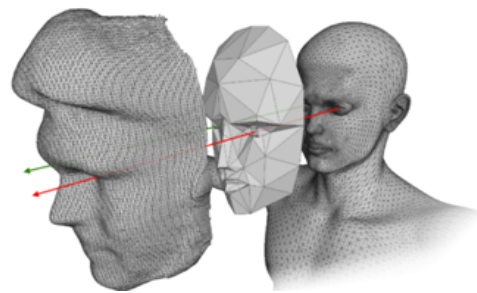


Figure 3: The pointcloud data is placed on the virtual avatar template with the help of the facetracking mesh.

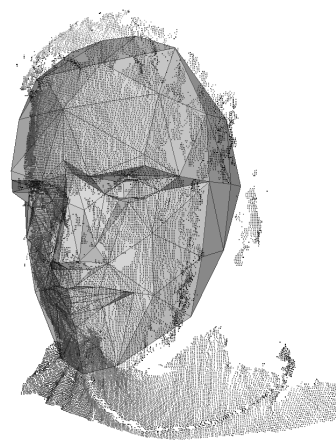


Figure 4: Pointcloud data from the Kinect sensor overlaid with the Candide 3 facetracking mesh.

4.1 Optimization

Since the creation of the virtual avatar is computationally complex, a fast implementation was needed. Early versions used the Kinect Fusion algorithm to obtain high quality depth maps. Our experiments came to the conclusion, that simple RGB-D images are sufficient.

We developed a fast C++ implementation using OpenCV and the EIGEN Library. The solver of the Eigen Library was utilized to solve the morphing equation. The Kinect SDK is used to capture the RGB-D frames and perform headtracking. The implementation has an interface to process live RGB-D frames from arbitrary cameras. It is possible to align the pointclouds and capture new frames at the same time in separate threads. For direct and fast display and further processing, there is an interface to retrieve the geometry data arrays in an OpenGL compatible format. An external obj file format importer/exporter was used to load the data from a Kinect capturing framework and to produce the output meshes.

5 Hair Reconstruction

Realistic virtual avatars need hair. There are various approaches, however only a few parameters can be obtained by the data from the Kinect sensor. The resolution of the Kinect RGB camera is not good enough to extract orientations of hair strands. Therefore only color, shape and geometry of the frontal hair are obtained.

To not completely rely on morphing, several hair templates for different length of hair were created. An algorithm detects each hair type by evaluating the color input images. The hair template as seen in Figure 11, is optimized by modifying the shape by scaling or morphing.

5.1 Texture atlas

Since several data frames are captured, all hair parameter detection can be performed on multiple frames and then averaged. To explore a reduction in the processing time, and eliminating false classifications, a texture atlas was created.

It was necessary to associate color information with each vertex of the pointcloud data. The vertices were projected into the corresponding texture. The pointcloud data from all frames was aligned in the same coordinate space. Just by removing the z-coordinate, the complete pointcloud data got projected into 2D. By interpolating the color information between the projected vertices, a texture atlas was obtained.

For the depth data, a texture atlas was created as well. Since the pointclouds are previously aligned in the same coordinate system, a simple plane projection was sufficient to create the depth-atlas. A depth and a RGB texture atlas can be seen in Figure 5.

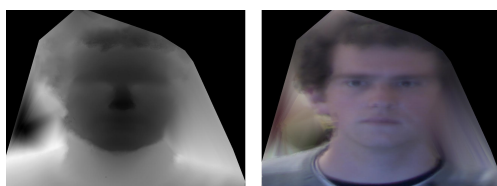


Figure 5: The texture atlas of the color and the depth images.

Performing all detections on all captured images proved to be more robust. Calculating the variance from the atlas for each image was a good method to eliminate erroneous frames. It was also possible to vote on the frames, and calculate a new weighted average texture atlas. Frames with a high similarity to the original texture atlas were assigned better weights than others. The improvement can be seen in Figure 6.

However, hair segmentation using the atlas turned out to be more difficult, because information is lost during the averaging process.



Figure 6: Original texture atlas on the left, weighted texture atlas on the right

5.2 Hair Segmentation

To procedurally generate a realistic hair mesh, information about the hair is required. Various methods were implemented to segment the hair on captured RGB images.

The first attempt was to segment hair with k-means clustering. By projecting the virtual avatar into the frame, the face could be masked. The trick of the k-means clustering in this application was to find a cluster containing hair. The area on the forehead above the hairline was masked. After the k-means clustering, the two most prominent clusters on forehead were selected. These clusters were likely hair clusters.



Figure 7: Red is the most prominent cluster, blue the second prominent cluster.

K-means clustering as seen in Figure 7, was not robust enough and did not consider the geometrical topology of the hair.

The Growcut algorithm [V105] as seen in Figure 8, takes sets of pixels of the image as input and performs iterative

clustering with cellular growth from these points. Pixels similar to the hair color were chosen for the hair as input. Random pixels in the face and on the background were defined as starting points for the face cluster.



Figure 8: Clustering with grow cut

Since these results were not good enough, segmenting the RGB images with a Graphcut implementation [GPS89] was evaluated. Even with the depth information and different color spaces, the results were not good enough.

Because the goal of this project was an applicable and robust implementation, a different strategy was chosen:

The hair length only gets estimated to choose a predefined hair template. The template later gets personalized by morphing. Only the following parameters had to be obtained from the RGB images

Hairline

To find the point where the hair starts on the forehead, the hairline needs to be estimated. The position of the facial features on the RGB images are known, since certain vertices just need to be projected into these images. The maximum magnitude of the color gradient between the eyebrows and the top of the head was searched on the 2D images. It turned out to be a very robust approximation of the hairline. The results of the hairline detection can be seen in Figure 9.

Hair color

A good approximation of the hair color was the average color of the pixels above the hairline. The pixels under the hairline were identified as skin color and used during the hair template morphing.



Figure 9: Red: Calculated hairline Green: Averaged hair-line parameter.

Hair length

The obtained hair length in this implementation is only a rough estimate to select a template with proper hair length. Three templates were created with Sculptris [Pix13]. The hair length was estimated efficiently by comparing the pixel color with the previously calculated hair color. Masking the face and the background helped a lot to get a robust estimate of the hair length. All parameters were obtained for every captured RGB-D frame separately and then averaged.

5.3 Hair Template Adaption

The hair length is classified into three categories. Depending on the detected category, a hair template mesh is loaded. Figure 10 shows some examples for possible hair templates. The template is positioned on the head of the virtual avatar as seen in Figure 12. The mesh gets projectively textured with a Kinect color image. Since the hair color is known, vertices with a different projected color are morphed.

All morphing operations face to the center of the head. The operation is applied per vertex, but the surrounding vertices are moved as well. The movement is linearly depended to the distance from the center vertex. This eliminates sharp creases and softens morphing. Several passes are performed. The projective texture is mapped again at the end of each pass, to ensure the morphing is stopped when the hair template is in shape. Figure 13 shows the result after the morphing.

Because there are only images of the front side of the head captured, the back of the head needs to be textured as well. To be able to also morph the back of the hair template, the back of the head was textured projectively with the front texture. However the face of the back texture was

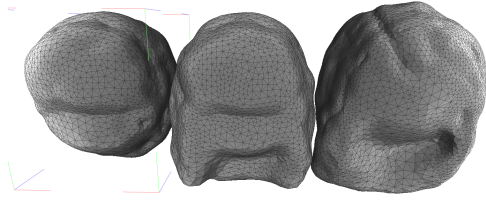


Figure 10: Hair templates for three different hair lengths.



Figure 11: Face mesh, face mesh with hair template, face mesh with adapted hair template

masked and the hair color was interpolated. As a side effect, the backside of the head also has a hair texture which blends with the front texture. The interpolation was simply done by replacing the face with the color of the hair as seen in Figure 14. There is room for improvement by using a better interpolation method.

6 Results

The results were retrieved in a scene with a clear background and bright lightning.

The final result as seen in Figure 15 consists of three textured meshes. The three meshes are the face, eyes and hair. They are separate meshes for exchangeability and to ease future animation. The meshes are saved in the .obj format, however the C++ implementation offers an interface to use the mesh data live and interactively.

6.1 Execution time

The time to generate a mesh from 7 RGB-D frames on a Pentium Dual Core CPU is about 45 seconds. However most of the computation time is needed by I/O operations with the slow OBJ fileformat. In an realistic application, the templates are previously loaded. Since the results do not have to be exported to an OBJ mesh, the system can work significantly faster. The most crucial and computationally intensive part of the implementation is solving the morphing equation. This part only takes 1.5 seconds using the solver of the EIGEN library.

6.2 Problems

One problem is the false morphing of the cheeks of the heads. In Figure 15, you can see white spots on some cheeks. This is caused by the lack of depth data in these

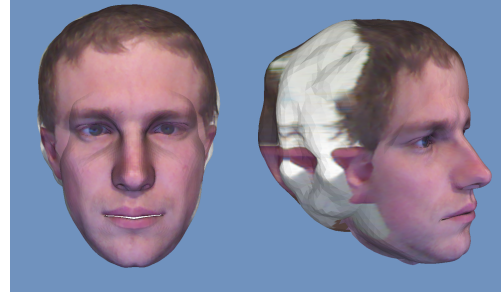


Figure 12: Virtual avatar with unmorphed hair template.



Figure 13: Virtual avatar with morphed hair.

regions. The RGB-D image are usually gathered just from the front side of the face. This might either be solved by removing certain morphing vectors, or reshaping the face mesh the same way as the hair mesh. Replacing the white spots with the skin color might also be a viable option.

Another problem is the low texture resolution on the side of the hair. This is a consequence of the dual projective texturing. Textures are projected on the front and back, but not on the sides. During the development of the implementation, using the best suiting texture for every polygon was evaluated, but the desired OBJ mesh export did not support multi-texture blending. However all the information can be obtained by the C++ interface, and a future interactive implementation can use shader programs to blend all RGB frames for optimal mesh texturing.

7 Conclusion

Automatic avatar generation can be used to enrich games and telepresence systems. We demonstrated that the concept of template adaptation can be extended to generate hair models and avatar generation can be performed in a time frame that is suitable for the named applications. Moreover, we came to the conclusion, that shortcomings of the depth sensor can be compensated well enough. However a bad RGB sensor yields directly to bad results. Our system therefore needs decent lighting and a contrasting neutral background. We assume that average users are capable of meeting these two requirements. Due to the wide availability of commodity depth sensors, such as the

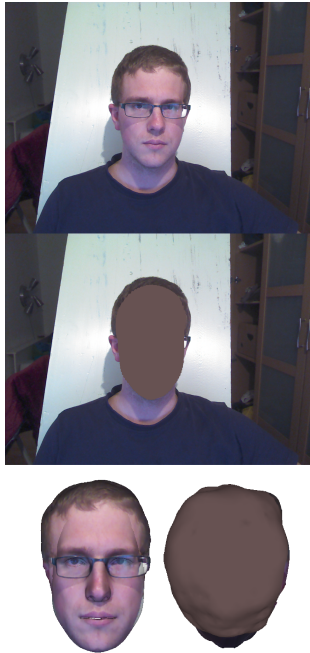


Figure 14: Texturing the backside. From top to bottom: Front texture, back texture with face automatically replaced, front and backside of the projectively textured mesh.

Microsoft Kinect, we believe that particularly games and applications will contain automatic avatar generation features in the future.

References

- [Ahl01] Jrgen Ahlberg. *CANDIDE-3 - An Updated Parameterised Face*. Tech. rep. 2001.
- [BV03] V. Blanz and T. Vetter. “Face recognition based on fitting a 3D morphable model”. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 25.9 (2003), pp. 1063–1074. ISSN: 0162-8828.
- [Cao+13] Chen Cao et al. “3D Shape Regression for Real-time Facial Animation”. In: *ACM Trans. Graph.* 32.4 (July 2013), 41:1–41:10. ISSN: 0730-0301.
- [Cha+13] Menglei Chai et al. “Dynamic Hair Manipulation in Images and Videos”. In: *ACM Trans. Graph.* 32.4 (July 2013), 75:1–75:8. ISSN: 0730-0301.
- [GPS89] D. M. Greig, B. T. Porteous, and A. H. Seheult. “Exact maximum a posteriori estimation for binary images”. In: (1989).
- [Iza+11] Shahram Izadi et al. “KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera”. In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST ’11. Santa Barbara, California, USA: ACM, 2011, pp. 559–568. ISBN: 978-1-4503-0716-1. DOI: 10.1145/2047196.2047270. URL: <http://doi.acm.org/10.1145/2047196.2047270>.
- [Jia+05] Dalong Jiang et al. “Efficient 3D reconstruction for face recognition”. In: *Pattern Recognition* 38.6 (2005). Image Understanding for Photographs, pp. 787–798. ISSN: 0031-3203.
- [Mag+13] AshrafY.A. Maghari et al. “Reconstructing 3D Face Shapes from Single 2D Images Using an Adaptive Deformation Model”. English. In: *Advances in Visual Informatics*. Ed. by HalimahBadioze Zaman et al. Vol. 8237. Lecture Notes in Computer Science. Springer International Publishing, 2013, pp. 300–310. ISBN: 978-3-319-02957-3.
- [Mic13] Microsoft. *Kinect Headtracking*. 2013. URL: <https://msdn.microsoft.com/en-us/library/jj130970.aspx>.
- [Pix13] Pixologic. *Sculptris*. 2013. URL: <http://pixologic.com/sculptris>.
- [Pro14] DAZ Productions. *DAZ 3D*. 2014. URL: <http://www.daz3d.com>.
- [Sum+13] E.A. Suma et al. “Rapid generation of personalized avatars”. In: *Virtual Reality (VR), 2013 IEEE*. Mar. 2013, pp. 185–185.

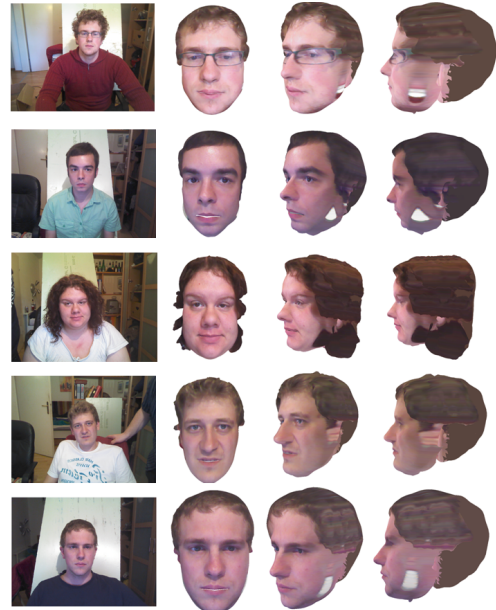


Figure 15: RGB Image and the final result.

- [V105] Vladimir Vezhnevets and et al. "*GrowCut*" – *Interactive Multi-Label N-D Image Segmentation By Cellular Automata*. 2005.
- [Zol+14] Michael Zollhofer et al. "Interactive model-based reconstruction of the human head using an RGB-D sensor". In: *Computer Animation and Virtual Worlds* 25.3-4 (2014), pp. 213–222. ISSN: 1546-427X.

Improved 3D Reconstruction using Combined Weighting Strategies

Patrick Stotko*

Supervised by: Tim Golla†

Institute of Computer Science II - Computer Graphics
University of Bonn
Bonn / Germany

Abstract

Due to the availability of cheap consumer depth sensors and recent advances in graphics hardware, scenes can be reconstructed in real time allowing a wide range of new applications. Current state-of-the-art approaches use a volumetric data structure and integrate recorded scans incrementally to provide complete and accurate reconstructions. However, scans usually contain noise and may be incomplete. Thus, using a simple update procedure becomes impracticable. To overcome these issues, we introduce a new weighting technique which combines different existing strategies. Typical strategies try to model such limitations, like varying visibility and depth-dependent noise, in order to estimate reasonable weights. Since the complexity of modeling grows extremely fast with respect to the number of considered limitations, development becomes complicated and prone to errors. Instead, we consider each limitation separately and construct easy-to-understand solutions for each one. Combining these small strategies leads to a more complex one and results in much higher quality reconstructions.

Keywords: Real-time Reconstruction, Voxel Hashing, GPU, Kinect, Weighting Strategies

1 Introduction

Since 3D models are extremely useful to describe the world, they are widely used in many different areas. Possible applications might be related to online stores. Buying furniture in big shops is often time-consuming and expensive. To save traveling costs and time, clients would like to use an interactive modeling tool for indoor rooms which allows detailed views of the new room from all possible perspectives. Other applications might be in the gaming industry. Modeling detailed real world objects manually is expensive and costs a lot of time and manpower. To reduce costs, already existing objects can be scanned in real time by cheap consumer hardware. In further steps, the given

3D model can be refined manually. Modeling large buildings is also quite expensive, so reconstructions captured by a drone can be very useful and increase work-flows.

For 3D reconstruction, several approaches are developed and many of them use the volumetric data structure by Curless and Levoy [5]. They partition the world into small voxels and store the reconstruction implicitly using a signed distance function (SDF). Thus, each voxel stores its signed distance from the estimated surface with respect to the camera. Since only the immediate region around the surface is actually needed, they also introduced the notion of the truncated signed distance function (TSDF). Instead of storing the exact distance to the surface, distances beyond the user-defined truncation region δ are cut off and only a relative one inside the interval $[-1, 1]$ is stored. Under this implicit scheme, scans can be integrated incrementally with a cumulative moving average. However, in terms of noisy data simple averaging is not appropriate since not every data point is equally useful for the reconstruction. So scan points need to be scored by a weight function.

The general problem of developing a weight function is its growing complexity. For each new considered aspect it increases by one dimension. Hence, development becomes error-prone and very slow. To overcome this issue, a strategy to reduce this complexity is needed.

In this paper, we present a new weighting technique which precisely addresses this issue. By separating the weight function into smaller ones, each desired aspect can be solved individually. This leads to easy-to-understand solutions which can be easily compared and discussed. To achieve a complete solution of the problem, we merge them together afterwards. As a result, this technique greatly reduces the problematic complexity and gives the user a powerful tool to customize the weights for his needs.

In the following sections, we first review current state-of-the-art approaches that provide different weighting strategies to solve sensor limitations. Then, we briefly introduce the structure of the 3D reconstruction algorithm used in this paper. Based on this algorithm, we present our new weighting technique and describe how it works with current strategies. Finally, we conclude by comparing them using our benchmark system.

*stotko@cs.uni-bonn.de

†golla@cs.uni-bonn.de

2 Previous work

3D reconstruction has been a very popular research topic in the last decades. Several different approaches were developed including point-based methods, height-map based representations and implicit volumetric approaches. In case of depth maps, one popular approach is the volumetric data structure introduced by Curless and Levoy [5]. It subdivides the world into voxels and stores scans using a signed distance function (SDF). To get a reconstruction, the surface is extracted using ray casting [6] or similar techniques. Their compelling results animated researches to develop several applications on top of this scheme.

One prominent application is KinectFusion [10, 11] which can reconstruct scenes in real time using the equally named Kinect sensor. Since it originally used a regular voxel grid, the space complexity was high and reconstructions were limited to small areas. To solve this drawback, several strategies are developed including moving volume approaches [16, 19, 20], streaming between CPU and GPU [3, 13] or efficient data structures on top of the regular voxel grid [3, 13, 15, 22]. One major improvement was achieved by Nießner et al. [13] who used a hash table to allow fast access to stored data. Voxel blocks, consisting of a set of typically 8^3 voxels, are managed through the hash table and efficiently processed in parallel. This technique achieved frame rates above the 30Hz frame rate of the Kinect. So we use this algorithm as a base and extend it with our new weighting strategy.

Other developments address the limitations of the depth sensor. Nguyen et al. [12] developed a weight function based on the measured noise of the Kinect sensor and achieved higher quality with much more details. Hemmat et al. [7, 8, 9] also presented a distance based function but only used scan points with higher weights for updating. With this approach, they achieved a similar reduction of the reconstruction error. Functions based on visibility are extensively analyzed by Sturm et al. [18]. Using them allows detailed 360° reconstructions and prevents invalid updates of voxels which causes strong artifacts.

3 Reconstruction algorithm

Since we have implemented the reconstruction algorithm of Nießner et al. [13] in CUDA [14], we start with a brief review of it. First, the given input frame is aligned to the current reconstruction to obtain the new camera pose. After estimation, we transform it into global coordinates and integrate it into the volume. In the last step, the new reconstruction is extracted and used for the next input frame.

3.1 Camera Pose Estimation

Before we can integrate the captured scan, the current pose of the camera has to be estimated. Typical approaches are based on the *Iterative Closest Point* algorithm (ICP)

[2, 4]. The idea is to find correspondences between two given point clouds and compute a rigid transformation $T = [R | t]$ which minimizes the point-to-plane error between the transformed source cloud P and the target cloud Q .

$$E = \sum_{k=1}^n \|(Tp_k - q_k) \cdot n_k\|_2^2 \quad (1)$$

Hence, finding robust correspondences is essential, so we use the efficient variant by Izadi et al. [10]. Correspondences are found using the given vertex and normal maps V_P, N_P and V_Q, N_Q by projecting each point $p_k \in P$ to image coordinates and choosing the point $q_l \in Q$ which projects to the same coordinates. If the distance between them and the angle between their normals is small, a match is found.

3.2 Integration

After the scan is transformed into global coordinates using the estimated transformation, it can be integrated into the volume. First, we determine all voxel blocks that fall into the current view frustum of the camera and integrate them into the hash table. This is performed by the GPU optimized variant of the *Digital Differential Analyzer* algorithm (DDA) [1] of Xiao et al. [21]. In the next step, we select all visible voxel blocks in the hash table and update them using the weighted cumulative moving average.

$$tsdf_{i+1} = \frac{tsdf_i \cdot w_i + tsdf \cdot w}{w_i + w} \quad (2)$$

$$w_{i+1} = w_i + w \quad (3)$$

Here, $tsdf_i$ and w_i are the old values of the voxel v , and $tsdf$ and w the TSDF value and weight estimated from the new depth map D_i . An appropriate choice for these estimated values is presented later (see section 4). After updating, outliers are removed through a garbage collection to keep the hash table sparse.

3.3 Surface Extraction

In the last step of the algorithm, the new reconstruction has to be extracted from the stored volume. For this task, we use ray casting [6]. First, we initialize the rays using the extrinsic and intrinsic parameters of our camera. Then, we compute the traversal intervals $[t_{start}(x, y), t_{end}(x, y)]$ for each output pixel $p = (x, y)$ by rasterizing all stored voxel blocks and generating two z-buffers.

If all these parameters are known, we sample the volume using *Adaptive sampling* [6]. The base step size t_0 is dynamically reduced to $t_1 = \frac{1}{8} \cdot t_0$ if the current distance to the surface is smaller than a threshold. To find immediately a point before and behind the surface during sampling, we set the step size t_0 to a multiple of the truncation region.

$$t_0 = c_{trav} \cdot \delta \quad (4)$$

This approach works best using values around $\frac{2}{3}$ since the traversal starts at the boundary of the truncation region

where the TSDF values are around ± 1 . At the end, we iteratively refine the found intersection and check whether this point is really a desired surface point.

$$|tsdf_{candidate} \cdot \delta| \leq c_{quality} \quad (5)$$

If its absolute distance to the stored reconstruction is smaller than a quality threshold $c_{quality}$, it will be accepted and returned. To get high quality results, typical values of this threshold are around one-hundredth of the voxel size. In a final step, we compose the extracted isosurface to the previous ones to get a full reconstruction. We summarize multiple reconstruction points with a voxel grid filter which has the same size as the infinite regular voxel grid of our volume.

4 Weighting strategies

Choosing appropriate weights is one of the most important parts during reconstruction since some scan points may be corrupted with noise or other limitations of the camera.

4.1 TSDF functions

The TSDF function describes a distance estimation from the reconstructed surface. Because we only search for the position in the volume with value 0 (see subsection 3.3), the real distance value sdf of a voxel to the surface is not needed and can be truncated.

KinectFusion [10, 11] provides a very simple TSDF function. It divides the real distance by the truncation region δ and cuts off large values.

$$tsdf_{KinFu}(sdf) = \begin{cases} -1 & \text{if } sdf < -\delta \\ \frac{sdf}{\delta} & \text{if } -\delta \leq sdf \leq \delta \\ 1 & \text{if } sdf > \delta \end{cases} \quad (6)$$

Voxels that are inside the truncation region are valued with truncated distances of the desired interval $[-1, 1]$, whereas those which are outside the region are valued with ± 1 .

Since this function may cause problems with noisy data, Nguyen et al. [12] introduced a function based on the noise of the Kinect. They modeled the noise along the view direction as a Gaussian distribution with zero mean and standard deviation σ_z which depends on the depth d of the measurement.

$$\sigma_z(d) = 0.0012 + 0.0019 \cdot (d - 0.4)^2 \quad (7)$$

The TSDF function is now given as a cumulative distribution function of the modeled noise distribution.

$$tsdf_{NM}(sdf, d) = \text{sign}(sdf) \sqrt{1 - e^{-\frac{2}{\pi} \frac{sdf^2}{\sigma_z(d)^2}}} \quad (8)$$

In contrast to KinectFusion, voxels near the boundary of the truncation region are also considered as far away.

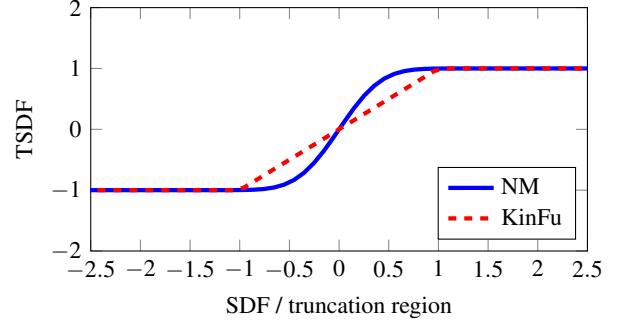


Figure 1: TSDF functions.

Therefore, finer details can be reconstructed since false zero-crossings due to noise are reduced.

For visualization in Figure 1, we fix the distance parameter d to 1.5 (which is the mean range of the Kinect) and use the relation $\delta = 3 \cdot \sigma_z(d)$ proposed by Nguyen et al. [12]. This suppresses the second dimension of the function and allows a visual comparison.

4.2 Weight functions

Weight functions are used to measure the importance of a scan point related to the currently updated voxel. In this paper, three different classes of strategies are discussed: Visibility based functions, depth based functions and angle based functions.

Visibility based functions This class of functions only uses SDF values to compute a weight. The signed distance implicitly encodes visibility information, so this class can be used in 360° reconstructions where a lot of occlusions have to be taken into account. The main drawback of them is depth-dependent noise. This effect can not be modeled here since only the relative distance of the voxel to the surface is known but not the absolute one.

In this context, the function of KinectFusion [10, 11] shows a natural behavior of scoring visibility.

$$w_{KinFu}(sdf) = \begin{cases} 1 & \text{if } sdf < 0 \\ \frac{sdf}{\delta} & \text{falls } 0 \leq sdf \leq \delta \\ 0 & \text{if } sdf > \delta \end{cases} \quad (9)$$

The function distinguishes between three different states. Voxels which lie before the measurement are scored with full weight of 1 indicating that they should be always updated. A similar scoring is performed for voxels lying behind the measurement. They might be part of the back side of the object or part of another one so updating them may cause problems. To prevent this, no update should be performed resulting in weights of value 0. The last state describes voxels inside the truncation region meaning that they lie close to the measured surface. Here, the weight decreases from 1 to 0 to ensure a smooth transition between the two other states.

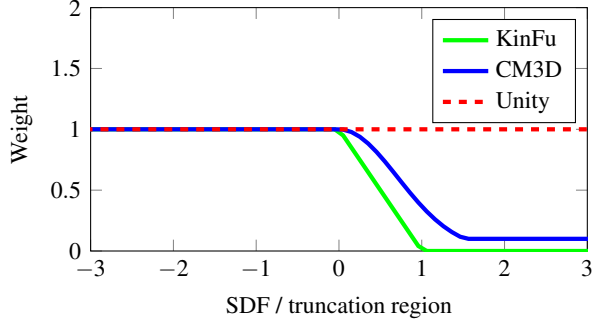


Figure 2: Visibility based weight functions.

However, skipping the update of the background can lead to reconstructions having a poor density of points and small holes on the back side (see subsection 5.2). To overcome this, Sturm et al. [18] developed a similar function.

$$w_{CM3D}(sdf) = \begin{cases} 1 & \text{if } sdf < 0 \\ \max\left(w_{min}, e^{-\frac{sdf^2}{\delta^2}}\right) & \text{if } sdf \geq 0 \end{cases} \quad (10)$$

Like KinectFusion, voxels in the foreground are updated with full weight of 1, while those lying inside the truncation region or far behind the measurement are scored with decreasing weights. But instead of a minimum weight of 0, a small positive one is used. This allows the algorithm to update the back side of the reconstructed object, while also keeping the original distance information. In this way, small holes are filled and the voxels can quickly be updated if better measurements become available. A visual comparison of both function is given in Figure 2.

Depth based functions In order to compute a weight, functions of this class use the absolute distance of the measured scan point to the camera. Hence, depth-dependent noise can be modeled here. But in contrast to visibility based functions, 360° reconstructions might be problematic since no occlusion information is provided.

As in their TSDF function, Nguyen et al. [12] use the depth to estimate the noise level first and then computes a reasonable weight.

$$w_{NM}(d) = \frac{\sigma_z(d_{min})}{\sigma_z(d)} \cdot \frac{d_{min}^2}{d^2} \quad (11)$$

$$\sigma_z(d) = 0.0012 + 0.0019 \cdot (d - 0.4)^2 \quad (12)$$

Here, $\sigma_z(d)$ is again the depth-dependent standard deviation of the modeled noise distribution. The weight itself is now computed as the quotient of the minimal noise and depth and the observed noise and depth. This leads to a strong decrease of confidence for voxels being far away from the camera. Additionally, Nguyen et al. [12] also encoded implicitly a visibility based function in their update process and used a 3×3 region around the desired pixel to reduce noise and fill small holes. Since the different classes

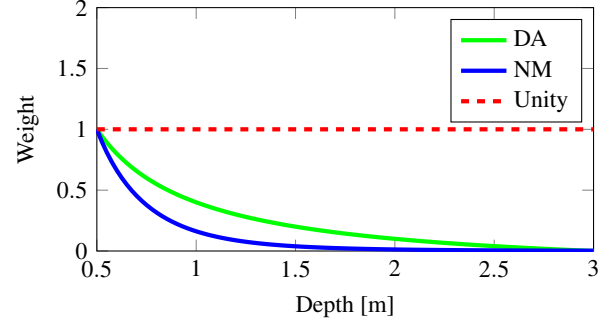


Figure 3: Depth based weight functions.

of weighting strategies are analyzed here, we drop them. The visibility based part is discussed separately and the modified update process using the region around the pixel only reduces noise perpendicular to the view direction of the camera which is not considered here.

A similar function modeling this kind of noise is developed by Hemmat et al. [7, 8, 9].

$$w_{DA}(d) = \frac{\frac{1}{d^2} - \frac{1}{d_{max}^2}}{\frac{1}{d_{min}^2} - \frac{1}{d_{max}^2}} \quad (13)$$

Like Nguyen et al. [12], larger distances are weighted with smaller weights where more noise is expected. The main difference between the two functions is the way how they handle the range interval $[d_{min}, d_{max}]$ of the camera. While Nguyen et al. [12] only use the minimal range as a normalization and ignores the upper bound, Hemmat et al. [7, 8, 9] use both bounds to compute weights between 0 and 1. Especially for larger intervals, the former scores two different high depth values with quite the same small weight, whereas the latter returns small weights which differ much more from each other to indicate the difference of the depth values.

Hemmat et al. [7, 8, 9] also used a modified update behavior. Instead of integrating all captured scans, only those with higher or similar weights than the previous ones are used. The idea here is to suppress measurements with higher noise than previously acquired ones to improve the quality. But as mentioned before, such modifications are not part of the considered class so we skip this step. For a better comparison in Figure 3, we also set the scaling parameter W_{max} in the original function of Hemmat et al. [7, 8, 9] to 1.

Angle based functions Another source of information about the noise level is contained in the angle between the surface normal and the view direction of the camera. Similar to the previously modeled distribution, noise increases significantly if this angle gets larger. Large angles indicate a relatively strong increase of the depth in the neighborhood of the desired pixel meaning that the surface is orientated away from the camera. In this case, measuring

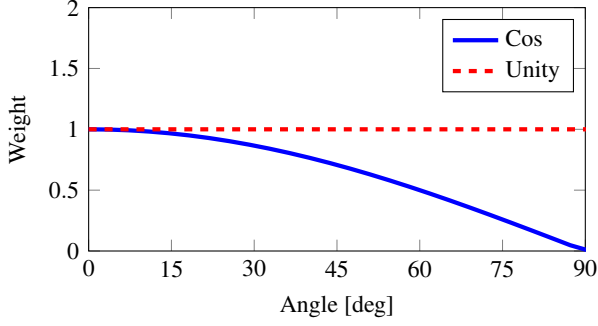


Figure 4: Angle based weight functions.

the correct depth becomes harder or even impossible, so a high noise level is expected here. To model this uncertainty, Curless and Levoy [5] use the cosine of the angle to compute weights.

$$w_{Cos}(\theta) = \cos \theta \quad (14)$$

As seen in Figure 4, high weights are assigned to voxels where the surface is orientated to the camera and the angle is small. With increasing angles, measurements become more and more inaccurate so the weights decrease. Angles around 90° indicate a rapid increase in depth and a high potential error. Measurements can be completely wrong now so no voxel should change its information and weights should be close to 0.

4.3 Combined functions

In the previous section, we introduced some easy to understand functions developed in the last past years. Researchers concentrated on several effects and achieved impressive results. However, these functions can not be easily extended. To overcome this issue, we separate them as described in the previous section by skipping terms that are not related to the desired class. The combination is now performed using a multiplication of the functions each chosen from a unique class.

$$w_{visibility} = f_{visibility}(sdf) \quad (15)$$

$$w_{depth} = f_{depth}(d) \quad (16)$$

$$w_{angle} = f_{angle}(\theta) \quad (17)$$

$$w_{combined} = w_{visibility} \cdot w_{depth} \cdot w_{angle} \quad (18)$$

One important aspect of a weight function is its bounds. We assume that all used functions are non-negative and bounded by some individual small constant.

$$\forall sdf: 0 \leq w_{visibility}(sdf) \leq W_{max\ visibility} \quad (19)$$

$$\forall d: 0 \leq w_{depth}(d) \leq W_{max\ depth} \quad (20)$$

$$\forall \theta: 0 \leq w_{angle}(\theta) \leq W_{max\ angle} \quad (21)$$

Since the combined values are bounded by the product of the individual bounds, these constants should be small.

This ensures that the combined function never reaches extremely large values. In our case, the combined bound is given by

$$W_{max\ combined} = W_{max\ visibility} \cdot W_{max\ depth} \cdot W_{max\ angle} \quad (22)$$

Ideally, all of them are set to 1 resulting in a combined bound of 1. Because large weights might be problematic in some implementations, future extensions like additional classes can now be integrated easily and the update behavior of the algorithm still remains as expected.

Naturally, also more than three strategies can be combined to compute weights. But using two functions of the same class does not necessarily increase the complexity of the entire strategy. The combination of these two functions is still a function of the same strategy class and is limited to its properties. So it could be defined directly without using this approach. However, this also can increase readability and lead to a finer grading of the modeled limitations.

5 Evaluation

In this part, we discuss the results achieved by our new technique and compare it with current approaches.

5.1 Test Environment

To allow applying the introduced algorithm and test the weight functions with existent 3D models, we need to compute depth maps on-the-fly. For construction, a virtual camera consisting of extrinsic and intrinsic parameters has to be defined first. The extrinsic parameters are also known as the pose and are already computed during pose estimation. The intrinsic parameters contain the camera resolution, its range and field of view. With these parameters, we compute the projection matrix introduced by Zhang [23].

$$P = \begin{pmatrix} \alpha & 0 & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{pmatrix} \quad (23)$$

The skewness parameter γ is dropped here since we only consider a virtual camera. The remaining coefficients can be expressed in terms of the camera resolution $n_x \times n_y$ and the horizontal field of view $fov_{horizontal}$.

$$\alpha = f \cdot u_0 = \beta \quad f = \frac{1}{\tan\left(\frac{fov_{horizontal}}{2}\right)} \quad (24)$$

$$u_0 = \frac{n_x}{2} \quad v_0 = \frac{n_y}{2} \quad (25)$$

To construct the depth map now, we first transform the given point cloud to the local camera coordinate system using the estimated pose. Then, we compute the distances to the camera and project all points by the projection matrix P . Points falling into the same pixel are summarized by the one with minimal distance to the camera. Finally, we store these distances in an image to obtain the final depth map.

		W_{single}						$W_{combined}$			
		Unity	KinFu	CM3D	NM	DA	Cos	KinFu + NM + Cos	KinFu + DA + Cos	CM3D + NM + Cos	CM3D + DA + Cos
Dragon	TSD F_{KinFu}	3.7052	1.9951	2.4717	3.0804	2.9989	3.3534	1.8656	1.8429	2.2461	2.2374
	TSD F_{NM}	3.3918	1.8711	2.4328	2.6763	2.5801	2.9604	1.7657	1.7434	2.2153	2.2042
Angel	TSD F_{KinFu}	2.9870	1.1662	1.5912	2.8978	2.9313	2.7380	1.1271	1.1290	1.5045	1.4977
	TSD F_{NM}	2.6760	1.1051	1.5117	2.5358	2.5328	2.4360	1.0835	1.0848	1.4385	1.4322

Table 1: Comparison of the mean error (ME) between different TSDF and weight functions (in mm).

		W_{single}						$W_{combined}$			
		Unity	KinFu	CM3D	NM	DA	Cos	KinFu + NM + Cos	KinFu + DA + Cos	CM3D + NM + Cos	CM3D + DA + Cos
Dragon	TSD F_{KinFu}	5.6066	2.9967	3.3495	4.6405	4.5039	5.1886	2.7912	2.7653	3.2000	3.2545
	TSD F_{NM}	5.2296	2.9073	3.3482	4.1469	3.9830	4.6493	2.7319	2.7061	3.1820	3.2213
Angel	TSD F_{KinFu}	4.5709	1.8169	2.1430	4.3310	4.3838	4.2221	1.7622	1.7800	2.0666	2.0574
	TSD F_{NM}	4.1803	1.7674	2.0573	3.8763	3.8563	3.8216	1.7310	1.7415	2.0097	1.9961

Table 2: Comparison of the root-mean-square error (RMSE) between different TSDF and weight functions (in mm).

For testing, we use the 3D models of the Asian Dragon and the Christian angel Lucy provided by the Stanford Computer Graphics Laboratory [17]. Reconstruction is performed on a Intel Core i7-4930K CPU, 32GiB RAM and a Nvidia GeForce GTX780 with 3 GiB VRAM. Our implementation uses a hash table with 2^{20} buckets each containing 2 entries. The reconstruction is stored in a predefined voxel buffer with a total number of 2^{18} voxel blocks and 8^3 voxels per block. We use a voxel size of 1 mm and a truncation region of 12mm. Our virtual camera has a range from 1.25 m to 2.25 m and captures depth maps using a resolution of 1920×1080 pixels with a horizontal field of view of 60° . Since the angel stands upright, the camera is rotated in this kind of situation and uses a resolution of 1080×1920 pixels with a vertical field of view of 60° . Additionally, we add artificial noise to each depth sample according to the previously mentioned Gaussian noise distribution with depth-dependent standard deviation $\sigma_z(d)$.

For reconstruction, all models are placed 1.75 m in front of the camera and scaled such that the heights of their bounding boxes are equal to 75% of the vertical height of the frustum at this depth. This ensures that the depth maps contains the full object independently of its original location in space.

5.2 Results

Reconstruction is performed by a single 360° round of the camera with 360 depth maps captured in total. Averaged reconstruction times per frame are 231.6ms (~ 4 fps), with 17.7ms (7.6%) for depth map creation, 89.8ms (38.8%)

for integration, 100.2ms (43.3%) for surface extraction and 23.9ms (10.3%) for surface composting. Since the used resolution is much higher than the one of the Kinect, this demonstrates the scalability of the volumetric data structure of Curless and Levoy [5].

Tests are performed among all possible combinations of TSDF and weight functions and shown in Table 1 and Table 2. More precisely, we test the unity function, the visibility based functions KinFu (Izadi et al. [10], Newcombe et al. [11]) and CM3D (Sturm et al. [18]), the depth based functions NM (Nguyen et al. [12]) and DA (Hemmat et al. [7, 8, 9]), and the angle based function Cos (Curless and Levoy [5]) as defined before. These functions only compute a single weight and are the reference of our technique. As the combined weights, we use all possible combinations consisting of three different functions each chosen from a unique class.

Because we consider a 360° reconstruction, visibility based functions perform best. KinectFusion seems to achieve the best result since it has the lowest error. However, the point density of its reconstruction is highly irregular. While there is a high density at the front side, the back side only has a low one resulting in a low total number of samples and many small holes. This is caused by using zero weight on the back side which means that no update is performed and relating voxel blocks are deleted by the garbage collection. Sturm et al. [18] use a small positive weight to overcome this and achieve very good results but with a slightly higher error. Depth based and angle based functions perform worse which is shown especially by the root-mean-square error that penalizes non-regular recon-

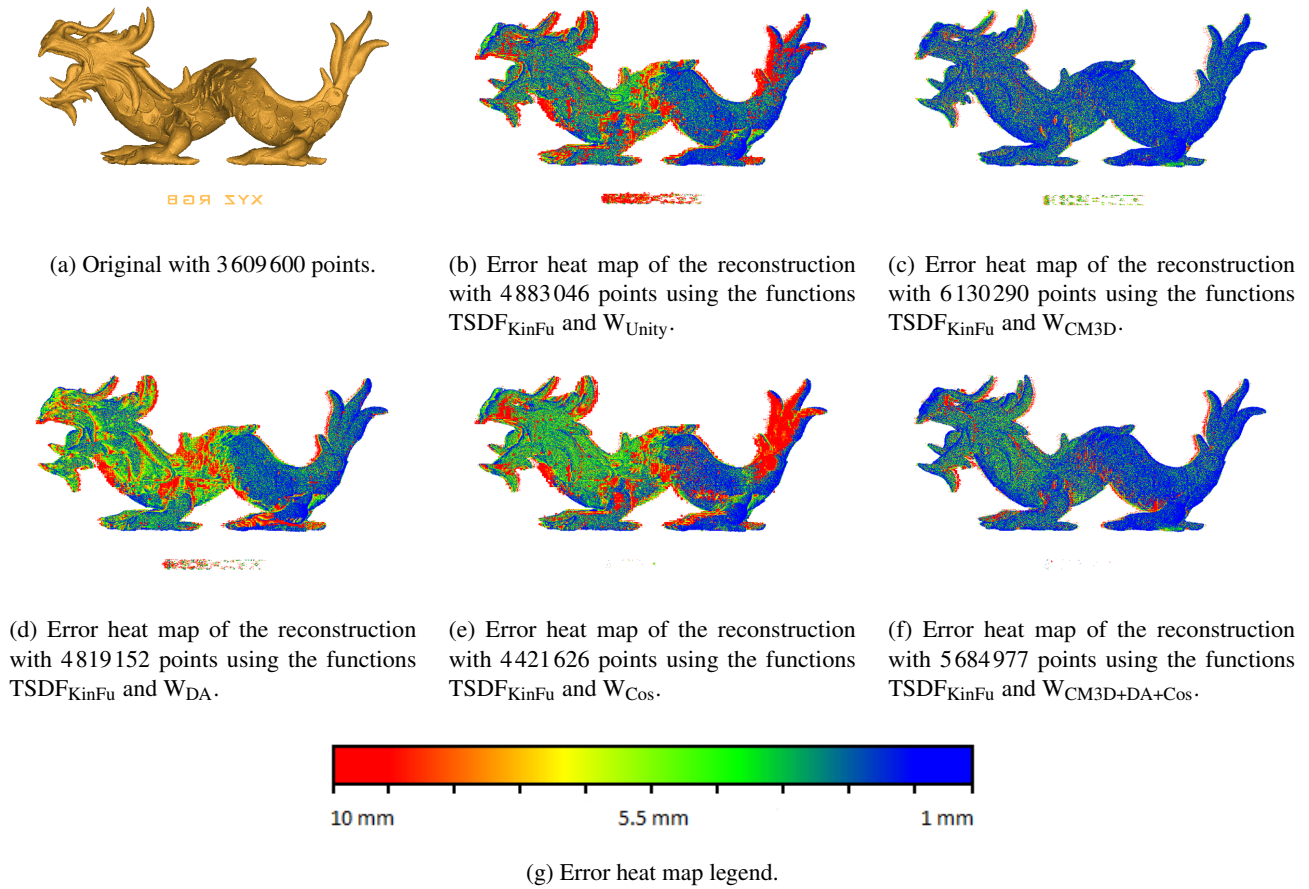


Figure 5: Asian dragon, used from Stanford Computer Graphics Laboratory [17].

structions. Furthermore, reconstructions are affected by many artifacts and only have a slightly better quality than the unity function (see Figure 5).

Best results among all test scenes are achieved by our new weighting technique. All combinations outperform the strategies which only use a single weight and achieve improvements up to 10%. Especially the combination with the function of Sturm et al. [18] performs best and achieves a lower reconstruction error than current approaches. However, not all regions of the reconstruction have a lower error. As shown in Figure 5, the depth based and angle based functions produce samples with high error at the head of the dragon while the visibility based ones create most samples with low error. As a result, the negative behavior of a function is also integrated in the overall strategy and can increase the error of some samples. Nevertheless, the positive and intended behavior of such a function outweighs its risks, so the averaged error over all samples is reduced.

6 Conclusion

We presented a new weighting technique which combines existing strategies and assigns them to a certain class that represents its properties. These classes can extend the

knowledge of the underlying problem and accelerate development of more sophisticated strategies. For a complete solution that captures all desired limitations, several functions are combined, each taken from a unique class.

We demonstrated high quality reconstructions with a smaller error than current state-of-the-art approaches. We believe that the advantages of combined functions are even more evident when more classes are developed and the complexity increases. However, combining arbitrary functions which might lower the error does not necessarily lead to better results. The functions we used are developed carefully and proven to perform good, so the combination of functions should also be chosen very carefully to achieve good results.

References

- [1] J. Amanatides and A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *In Eurographics '87*, pages 3–10, 1987.
- [2] P. J. Besl and N. D. McKay. A Method for Registration of 3-D Shapes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(2):239–256, 1992.

- [3] J. Chen, D. Bautebach, and S. Izadi. Scalable Real-time Volumetric Surface Reconstruction. *ACM Trans. Graph.*, 32:113:1–113:16, 2013.
- [4] Y. Chen and G. Medioni. Object Modelling by Registration of Multiple Range Images. *Image Vision Computing (IVC)*, 10(3):145–155, 1992.
- [5] B. Curless and M. Levoy. A Volumetric Method for Building Complex Models from Range Images. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 303–312, 1996.
- [6] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. In *Computer Graphics Forum*, volume 24, pages 303–312, 2005.
- [7] H. J. Hemmat, E. Bondarev, and P. H. N. de With. Exploring Distance-Aware Weighting Strategies for Accurate Reconstruction of Voxel-Based 3D Synthetic Models. In *MultiMedia Modeling - 20th Anniversary International Conference, MMM 2014, Dublin, Ireland, January 6-10, 2014, Proceedings, Part I*, pages 412–423, 2014.
- [8] H. J. Hemmat, E. Bondarev, G. Dubbelman, and P. H. N. de With. Improved ICP-based Pose Estimation by Distance-aware 3D Mapping. In *VISAPP 2014 - Proceedings of the 9th International Conference on Computer Vision Theory and Applications, Volume 3, Lisbon, Portugal, 5-8 January, 2014*, pages 360–367, 2014.
- [9] H. J. Hemmat, E. Bondarev, G. Dubbelman, and P. H. N. de With. Evaluation of Distance-Aware KinFu Algorithm for Stereo Outdoor Data. In *VISAPP 2014 - Proceedings of the 9th International Conference on Computer Vision Theory and Applications, Volume 2, Lisbon, Portugal, 5-8 January, 2014*, pages 746–751, 2014.
- [10] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and A. Fitzgibbon. KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, pages 559–568, 2011.
- [11] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-Time Dense Surface Mapping and Tracking. In *IEEE ISMAR*, 2011.
- [12] C. V. Nguyen, S. Izadi, and D. Lovell. Modeling Kinect Sensor Noise for Improved 3D Reconstruction and Tracking. In *3DIMPVT*, pages 524–530, 2012.
- [13] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3D Reconstruction at Scale Using Voxel Hashing. *ACM Trans. Graph.*, 32(6):169:1–169:11, 2013.
- [14] Nvidia. CUDA. http://www.nvidia.com/object/cuda_home_new.html, 2007. Accessed March 31, 2015.
- [15] F. Reichl, M. G. Chajdas, K. Bürger, and R. Westermann. Hybrid Sample-based Surface Rendering. In *Vision, Modeling and Visualization*, pages 47–54, 2012.
- [16] H. Roth and M. Vona. Moving Volume KinectFusion. In *Proceedings of the British Machine Vision Conference*, pages 112.1–112.11, 2012.
- [17] Stanford Computer Graphics Laboratory. The Stanford 3D Scanning Repository. <http://graphics.stanford.edu/data/3Dscanrep/>, 1996. Accessed March 31, 2015.
- [18] J. Sturm, E. Bylow, F. Kahl, and D. Cremers. CopyMe3D: Scanning and Printing Persons in 3D. In *German Conference on Pattern Recognition*, 2013.
- [19] T. Whelan, H. Johannsson, M. Kaess, J. Leonard, and J. McDonald. Robust Tracking for Real-Time Dense RGB-D Mapping with Kintinuous. Technical report, Computer Science and Artificial Intelligence Laboratory, MIT, 2012.
- [20] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, and J. McDonald. Kintinuous: Spatially Extended KinectFusion. In *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, 2012.
- [21] K. Xiao, D. Z. Chen, X. S. Hu, and B. Zhou. Efficient implementation of the 3D-DDA ray traversal algorithm on GPU and its application in radiation dose calculation. *Medical Physics*, 39(12):7619–7625, 2012.
- [22] M. Zeng, F. Zhao, J. Zheng, and X. Liu. A Memory-Efficient Kinectfusion Using Octree. In *Proceedings of the First International Conference on Computational Visual Media*, pages 234–241, 2012.
- [23] Z. Zhang. A flexible new technique for camera calibration. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(11):1330–1334, 2000.

3D Reconstruction of Buildings from LiDAR Data

Marko Bizjak*

Supervised by: Domen Mongus†

Laboratory for Geometric Modelling and Multimedia Algorithms
University of Maribor Faculty of Electrical Engineering and Computer Science
Smetanova ulica 17, SI-2000 Maribor / Slovenia

Abstract

This paper presents a new approach to automatic 3D building reconstruction from LiDAR data. While traditional approaches use random sampling or Hugh transform for extracting subsets of coplanar points from noisy point clouds, our method is based on locally fitted surfaces (LoFS). These are planes, best-fitted to the K-neighbourhood of each LiDAR point. In this way, a set of candidate patches for a building surface is obtained. The clustering of patches is then performed based on the planes' normals and the positions of neighbourhoods, in order to obtain a rough approximation of flat roof sides. An adjacent graph is generated between them and intersections between neighbouring sides are estimated in order to define ridges, while intersections between buildings and ground points are considered in footprint definition. This defines the vertical walls. This method was tested on buildings of different architectural styles, sizes, and complexity. Most buildings are successfully reconstructed, however with increased building details, the accuracy of reconstruction is often decreased.

Keywords: LiDAR, Building, Reconstruction, LoFS

1 Introduction

Light Detection And Ranging (LiDAR) has become a popular research topic over the last decade, as more attention is directed towards Earth observations [10, 9]. LiDAR is an active remote sensing technology that utilises laser light in order to scan surface topographies, usually from an airborne platform. The result of such scanning is a dense cloud of topologically unstructured 3D points that allows accurate monitoring of the Earth's surface. Recently, 3D reconstruction of urban environment has become increasingly important and is being used for many applications such as urban planning [4], wireless communications' modelling [18], tourism or grand-scale virtual geographical information programs [19]. As manual reconstruction is exhausting, it is imperative to be able to reconstruct buildings automatically or semi-automatically.

This paper presents a novel method for automatic building reconstruction from LiDAR data that uses locally fitted surfaces (LoFS) in combination with clustering and an adjacency graph to find primary buildings' vertexes and borders. The paper is structured into 4 sections. The next section provides an overview of the related work. Section 3 describes the proposed method for reconstruction of buildings, followed by the results. The last section concludes this paper.

2 Related work

Automatic reconstruction of buildings from LiDAR data is an intensive research field, where a number of solutions have already been proposed. In most common cases, planar patches are extracted from point clouds in order to obtain approximated flat roof sides. Random sampling consensus (RANSAC) [2, 21, 22, 12, 1], Hugh transform [13, 20, 23], or region growing on surfaces [15, 7, 5, 17] are the most often used methods for this purpose.

An early attempt at semi-automatic reconstruction was done by Haala and Brenner [3]. In addition to LiDAR data they used 2D ground plans of buildings for their automatic 3D reconstruction. The ground plans are divided into rectangles, for each of which 3D primitives are instantiated. Final reconstruction is obtained by merging selected 3D primitives. Later, Brenner [2] presented a bottom-up approach that extracts faces from laser scan data using RANSAC. A set of rules was developed to decide which segments are selected for this purpose. The roof is then built from the selected segments, closing any gaps. Recently, Arikan et. al [1] introduced a reconstruction and modelling pipeline to create polygonal models from unstructured point clouds. They extracted planar patches using RANSAC and then snapped them together using an iterative optimisation approach.

In contrast, Vosselman [23] developed a method that uses Hough transform in order to extract planar faces from laser scan data, followed by a connected component analysis. The roof topology is determined by considering geometric constraints and bridging gaps along detected edges. Vosselman and Dijkman [24] upgraded this method by integrating the information obtained from ground plans.

*m.bizjak@um.si

†domen.mongus@um.si

In 2003, region growing on surfaces was utilised for reconstruction by Rottensteiner and Breise [17]. They proposed a method that uses LiDAR data in combination with aerial images. Roof planes are detected by a curvature-based segmentation technique [16]. They are grouped to create polyhedral building models and then improved with the usage of all available sensor information. A region growing algorithm based on an adjacency graph was proposed by Milde et al. [7]. Simple roof shapes are extracted by finding subgraphs, whereas complex roof structures are derived using formal grammar.

The idea of presenting topological relations between approximated roof sides using adjacency graph was first introduced by Verma et al. [22], where two faces are considered as adjacent if at least one pair of line segments from their approximated 3D boundaries is close enough. Several other approaches have also been proposed for establishing adjacency relations. Milde et al. [7] used the perpendicular distance between oriented parallel bounding boxes of faces. Oude Elberink [14] considered the length of a segment, determined by points within a flexible distance from intersection lines between two faces.

3 Reconstruction of buildings

Reconstruction is performed over four steps. Firstly, locally fitted surfaces (LoFS) are estimated for each point. In the next step DBSCAN clustering is applied on the normals and position of the neighbourhood in order to obtain approximated flat roof sides. An adjacency graph is constructed in the third step and in the last step main buildings' borders and sides are estimated. Every step is separately described in detail in the following subsection.

3.1 Estimation of LoFS

The method's input is a LiDAR point cloud, where each point is georeferenced and classified as building, terrain or vegetation [8]. Firstly, for each building point, the K-nearest points are located using fast approximate K-nearest neighbours algorithm [11]. A plane is fitted to the K-neighbourhood of each point using locally fitted surfaces (LoFS). LoFS is a set of best-fitted surfaces to the K-neighbourhood of each point [8]. If the neighbourhood of each point does not belong to the same surface, large-fitting error occurs. In this case, a surface with better-fit should exist. In order to determine it, the neighbourhood of each point is inspected. For better understanding, consider the example in Figure 1a), where a case of six points from a roof surface and a point within the building is presented. Firstly, a set of best-fitting surfaces is estimated in Figure 1b) with a fitting window size set to 3 and linking window size to 5. Window size defines the number of points considered when fitting or linking surfaces. In our case this means that each surface is fitted to a given point by also considering its neighbour on each side. Ev-

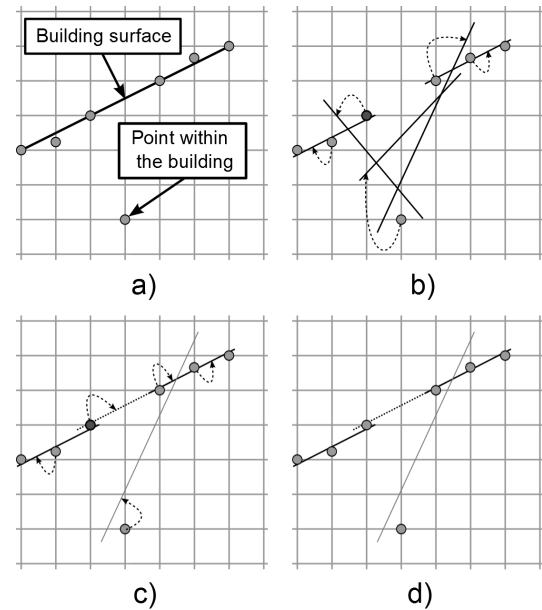


Figure 1: Estimation of LoFS with window size for fitting set at 3 and for linking at 5 in the case of a noisy surface a). A set of best-fitted surfaces is obtained and linked with the corresponding points b). The neighbourhood of each point is inspected to link it with a surface with the best fit c) in order to obtain the final set of surfaces d).

ery point is linked to the fitted surface, as shown in Figure 1b). During the linking step, the neighbourhood of 2 points on each side of the given point are inspected for the defined window size. The surface from this neighbourhood with the lowest distance (error) to a given point is linked with it. Thus, the darkened point is linked with the surface corresponding to the second neighbour on its right (as shown in Figure 1c)). The final set of surfaces can be seen in Figure 1d).

3.2 Clustering

Points linked with their LoFS are then clustered separately by normals and then by position in order to obtain a rough approximation of flat roof sides. Clustering is a process of dividing data into groups of similar objects (clusters). Density based spatial clustering (DBSCAN) within large datasets with noise is used [6], as LiDAR point cloud is a representative of such datasets. DBSCAN is based on the idea that the density within a neighbourhood for an object has to be high enough to belong to a cluster. Each cluster is created from a single data object by absorbing all objects in its neighbourhood. DBSCAN is independent of data order. It is controlled by two parameters: the minimal number of points required to be considered as a cluster and density threshold for neighbourhoods. In this step clusters of points that belong to the same flat roof side are obtained. The problem of using density within a neighbourhood for clustering is that DBSCAN also clusters points within the

neighbourhood that might be a part of a curved surface. This only occurs when the curvature is small enough for the distance between neighbours to be lower than the density threshold. Each cluster with an averaged plane equation is considered as a node of the adjacency graph that is constructed in the next step. The result of such clustering is presented in Figure 2.



Figure 2: Example of clusters of points that belong to the same flat roof side (i. e. nodes) for a building's roof.

3.3 Adjacency graph construction

The topological relations between approximated roof sides are usually presented in an adjacency graph. Adjacency is commonly defined as a pair-wise connection by an edge between two roof sides that share a common border [22]. In order to be able to detect common ridge points, we define adjacency as edges between roof sides that share at least one ridge point. Adjacency is tested using enlarged and oriented bounding boxes of the approximated roof sides. If two bounding boxes overlap, they are adjacent. In this way an undirected graph is constructed, as shown in Figure 3.

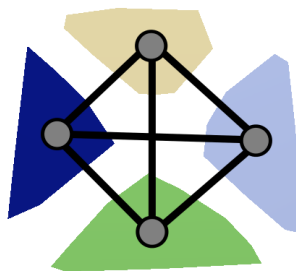


Figure 3: Adjacency graph for a roof with four nodes that share the same ridge point. Black segments represent graph edges.

3.4 Modelling

In order to reconstruct a building we need to determine its boundary points and edges. Firstly, ridge points that are shared between at least three roof sides are estimated. This

is performed by searching for maximal cliques in the adjacency graph. Clique is a subset of nodes in an undirected graph where every two nodes in the subset are connected by an edge. It is maximal when it does not exist within a larger clique. From every clique we select three nodes that share at least one border with the other two nodes. To test if two nodes share a common border, an intersection line between pairs of nodes is first calculated. Then the points of each node that are within a certain distance d from the intersection line are projected perpendicularly on the line, as shown in Figure 4.

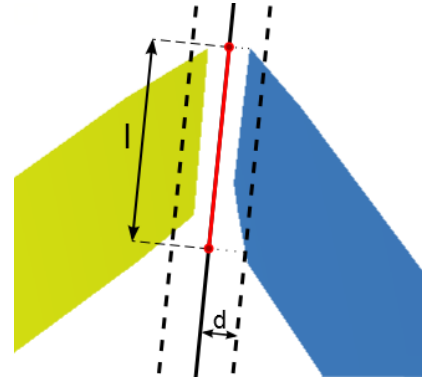


Figure 4: Shared common border between a pair of nodes. Points within a distance d from both nodes are perpendicularly projected on the intersection line. The longest segment between projected points l needs to be long enough.

If the longest segment l between the projected points from both nodes is long enough (e.g. 1m), it is considered that there is a border between these nodes. Using the selected three nodes we calculate a shared ridge point as the intersection point of three planes. The calculated ridge point is shared amongst all nodes in the clique. After we have obtained ridge points, the borders between nodes are estimated. A border is a segment on an intersection line between two nodes. A segment is bounded by either a ridge point or the projection of a bounding point of a node. There are two types of bounding points of a node used for projection.

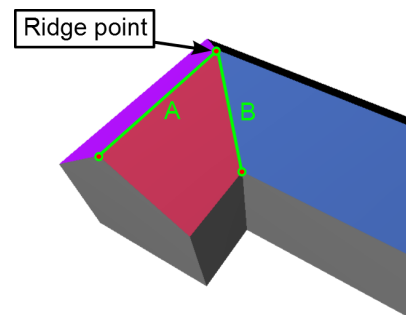


Figure 5: Border bounding points determination based on the type of a border (A - horizontal, B - inclined).

Suppose we have a building designed as shown in Figure 5, we already know the marked ridge point and we need to determine borders A and B. If an intersection line is horizontal (A), it represents a ridge, thus the bounding point of a border is the perpendicular projection of a node point to the intersection line in such a way that the border is maximally long. If it is not horizontal (B), then the bounding point of a border is the point on the intersection line with the same height as for the lowest point from both intersecting nodes. After the bounding points and borders of a roof are determined, we need to determine the building's exterior walls' height. This height is obtained from LiDAR data as the difference to the lowest ground point in direct proximity to the building.

4 Results

The presented method was tested on LiDAR datasets with a wide variety of building types. The tested buildings were of different sizes, architecture and complexity. We were limited by LiDAR data sparsity as the number of points on each planar surface needs to be large enough for successful extraction of flat roof sides. Consequently, only those flat roof sides that are large enough were successfully extracted and used during the process of reconstruction.

During the first step we fitted to the neighbourhood of 8 points ($K=8$). The fitting and linking window sizes were also set to 8. At least 5 points were needed to form a cluster in the next step. When clustering by position and normals were performed, densities of 1.4 and 0.12 were used, respectively. In the third step for the adjacency test the oriented bounding boxes were enlarged by 1m in all directions. For the shared common border test d and the minimum length of l were both set to 1m. All the presented results were tested using these settings. The result of the reconstruction of a single building without complex parts is presented in Figure 6.

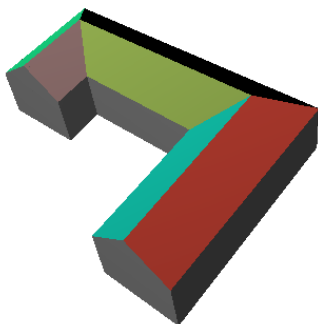


Figure 6: Reconstruction of a building with large surfaces.

Common ridge points and borders were successfully determined. Additionally we are able to reconstruct more complex buildings with smaller roof surfaces, as shown in

Figures 7 and 8. For better comparison, LiDAR datasets for these two reconstructed buildings are shown in Figures 7a) and 8a), where the building points are red and ground points brown. The capabilities of reconstruction include a hipped roof and embedding gable to the larger roof sides, as can be seen in Figure 7.

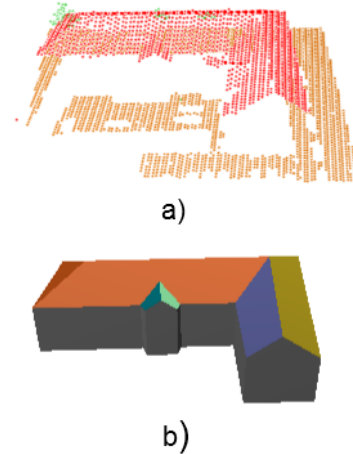


Figure 7: Reconstruction of a building b) with embedded gable and hipped roof on the left side from LiDAR data a).

Figure 8 presents an example of a reconstruction that incorporates shed dormer into the building. Dormers can be located anywhere on the larger roof surface as long as the first two steps have successfully extracted its planar surfaces. In addition we performed testing on larger Li-

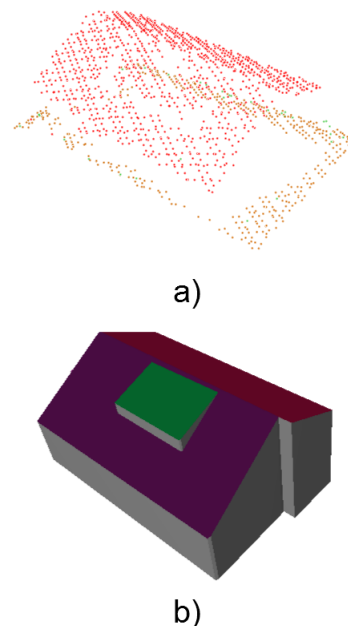


Figure 8: Reconstruction of a building b) with embedded shed dormer from LiDAR data a).

DAR datasets. As can be seen in Figures 9 and 10, this method provides good reconstruction of buildings from LiDAR datasets with a greater number of buildings.

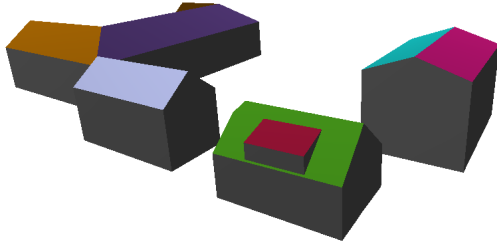


Figure 9: Example of successfully reconstructed buildings of a small settlement from LiDAR dataset.

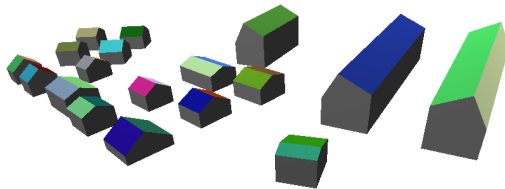


Figure 10: Reconstruction of a settlement.

5 Conclusions

This paper proposed a novel method for the reconstruction of buildings from LiDAR data. For the extraction of approximated roofs' planar faces the locally fitted surfaces (LoFS) and DBSCAN clustering were used. Between the obtained planar faces an adjacency graph was constructed for extracting the common ridge points. The borders between the faces and building's exterior walls were estimated for the final building model. To our knowledge this is the first method using this concept of LoFS for the estimation of planar surfaces. The results confirmed that the method can successfully reconstruct most regularly complex buildings with sufficient accuracy.

There are many possibilities for the improvement of reconstruction during all steps. Different clustering algorithm could provide better planar faces extraction results as DBSCAN clusters points within the neighbourhood that might also be a part of a curved surface. For faster computation a subgraph of the adjacency graph from the third step, that would define adjacency more strictly, could be used for border estimation. Modelling improvements are possible on many levels such as multi-layered building roofs, curved roofs or facade design.

References

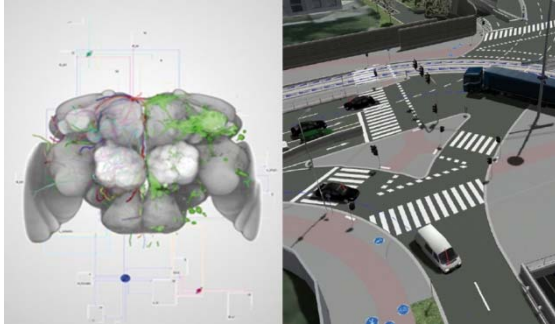
- [1] M. Arikan, M. Schwärzler, S. Flöry, M. Wimmer, and S. Maierhofer. O-snap: Optimization-based snapping for modeling architecture. *ACM Trans. Graph.*, 32(1):1–15, 2013.
- [2] C. Brenner. Towards fully automatic generation of city models. *International Archives of Photogrammetry and Remote Sensing*, 33(B3/1):84–92, 2000.
- [3] C. Brenner and N. Haala. Rapid acquisition of virtual reality city models from multiple data sources. *International Archives of Photogrammetry and Remote Sensing*, 32:323–330, 1998.
- [4] J. Döllner, T. H. Kolbe, F. Liecke, T. Sgouros, and K. Teichmann. The virtual 3D city model of Berlin—managing, integrating, and communicating complex urban information. In *Proceedings of the 25th Urban Data Management Symposium UDMS*, 2006.
- [5] P. Dorninger and N. Pfeifer. A comprehensive automated 3D approach for building extraction, reconstruction, and regularization from airborne laser scanning point clouds. *Sensors*, 8(11):7323–7343, 2008.
- [6] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *KDD*, 96(34):226–231, 1996.
- [7] J. Milde, Y. Zhang, C. Brenner, L. Plmer, and M. Sester. Building reconstruction using a structural description based on a formal grammar. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 37:227–232, 2008.
- [8] D. Mongus, N. Lukač, and B. Žalik. Ground and building extraction from LiDAR data based on differential morphological profiles and locally fitted surfaces. *ISPRS Journal of Photogrammetry and Remote Sensing*, 93:145 – 156, 2014.
- [9] D. Mongus and B. Žalik. Parameter-free ground filtering of LiDAR data for automatic DTM generation. *ISPRS Journal of Photogrammetry and Remote Sensing*, 67:1–12, 2012.
- [10] D. Mongus and B. Žalik. Computationally efficient method for the generation of a digital terrain model from airborne LiDAR data using connected operators. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 7(1):340–351, 2014.
- [11] M. Muja and D.G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.

- [12] L. Nan, A. Sharf, H. Zhang, D. Cohen-Or, and B. Chen. Smartboxes for interactive urban reconstruction. *ACM Trans. Graph.*, 29(4):1–10, 2010.
- [13] A. Novacheva. Building roof reconstruction from LiDAR data and aerial images through plane extraction and colour edge detection. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 37:53–57, 2008.
- [14] S. Oude Elberink. Target graph matching for building reconstruction. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 38:49–54, 2009.
- [15] J. Park, I. Lee, Y. Choi, and J. Y. Lee. Automatic extraction of large complex buildings using LiDAR data and digital maps. *International Archives of Photogrammetry and Remote Sensing*, 36:148–154, 2006.
- [16] F. Rottensteiner and C. Bries. A new method for building extraction in urban areas from high-resolution LiDAR data. *International Archives of Photogrammetry Remote Sensing and Spatial Information Sciences*, 34(3/A):295–301, 2002.
- [17] F. Rottensteiner and C. Bries. Automatic generation of building models from LiDAR data and the integration of aerial images. *Int. Arch. of Photogrammetry and Remote Sensing*, 34:174–180, 2003.
- [18] T.K. Sarkar, Zhong Ji, Kyungjung Kim, A. Medouri, and M. Salazar-Palma. A survey of various propagation models for mobile communication. *Antennas and Propagation Magazine, IEEE*, 45(3):51–82, 2003.
- [19] S. Sheppard and P. Cizek. The ethics of Google Earth: Crossing thresholds from spatial data to landscape visualisation. *Journal of Environmental Management*, 90(6):2102 – 2117, 2009.
- [20] G. Sohn, X. Huang, and V. Tao. Using a binary space partitioning tree for reconstructing polyhedral building models from airborne LiDAR data. *Photogrammetric Engineering and Remote Sensing*, 2008.
- [21] F. Tarsha-Kurdi, T. Landes, and P. Grussenmeyer. Extended RANSAC algorithm for automatic detection of building roof planes from LiDAR data. *The photogrammetric journal of Finland*, 21(1):97–109, 2008.
- [22] V. Verma, Rakesh Kumar, and S. Hsu. 3D building detection and modeling from aerial LiDAR data. 2:2213–2220, 2006.
- [23] G. Vosselman. Building reconstruction using planar faces in very high density height data. *IAPRS*, 32(3):87–92, 1999.
- [24] G. Vosselman and S. Dijkman. 3D building model reconstruction from point clouds and ground plans. 34(3/W4):37–44, 2001.

Sponsors of CESC 2015



zentrum für
virtual reality und visualisierung
forschungs-gmbh



VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH

The VRVis Research Center is a joint venture in research and development for virtual reality and visualization. VRVis was founded in 2000 as part of the Austrian Kplus program to bridge the gap between academic research and commercial development as well as to supply the necessary transfer of knowledge between the academic community and industry. The competence center VRVis is funded by BMVIT, BMWFW, and ZIT - The Technology Agency of the City of Vienna within the scope of COMET – Competence Centers for Excellent Technologies. The program COMET is managed by FFG.

This mission is mirrored in a variety of academic and industrial partners. The research center is currently conducted by five academic institutes and numerous industrial partners. Leading-edge innovations and down-to-earth business style characterizes VRVis as a valued partner for high-level research.

The company is located in Vienna, Austria. Today, around 60 researchers together with about 20 students do high-level applied and basic research in three different areas.

The Team

VRVis consists of internationally experienced researchers in the areas of visualization, rendering and visual analysis. Their outstanding experience and knowledge in these topics qualify them for the innovative research they are performing. The research areas are headed by key researchers who manage these areas, define goals and projects for this area, and conduct the defined research together with their staff. All members of the research team are young researchers, whose creativity and ingenuity is the key to the success. VRVis is always looking for young, talented, and motivated researches in the fields of research to extend its research work or to support partner companies.

Research Program

The scientific research program consists of three research areas (Visualization, Rendering and Visual Analysis) in which thematically matching research projects are conducted. Each research area realizes application projects on the one hand and basic research for these application projects on the other hand.

Working at VRVis

VRVis is always looking for students, junior and senior researchers who want to join the VRVis team. VRVis is offering internships, diploma theses and PhD theses in cooperation with the TU Wien and regular positions. For more information or search for job opportunities in the field of Visual Computing visit our webpage at www.vrvis.at.

Selection of Partners

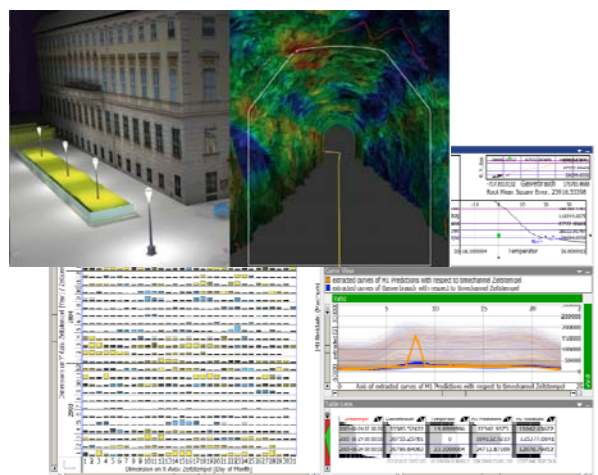
Scientific Partners:

- Vienna University of Technology
- Graz University of Technology
- University of Vienna

Industrial Partners:

- AVL List GmbH
- AGFA Healthcare GesmbH
- Austria Power Grid AG
- Geodata Ziviltechniker GmbH
- Imagination Computer Services GesmbH
- ÖBB-Infrastruktur AG
- Zumtobel Lighting GmbH
- and many more

Currently, VRVis is again extending its industrial base with new partners from several new fields.



Additional Information and Contact

Please visit our webpage for detailed information about the research program or current projects at www.vrvis.at or contact us at office@vrvis.at or via phone +43 (1) 20501 / 30100.

